Google Cluster Computing Faculty Training Workshop

Module IV: MapReduce Theory, Implementation, and Algorithms

This presentation includes content © University of Washington and/or Google, Inc.

Redistributed under the Creative Commons Attribution 3.0 license.

All other contents:
© Spinnaker Labs, Inc.

Overview

- Functional Programming Recap
- MapReduce Theory & Implementation
- MapReduce Algorithms

© Spinnaker Labs, Inc.

Functional Programming Review

- Functional operations do not modify data structures: They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter

© Spinnaker Labs, Inc.

Functional Programming Review

```
fun foo(l: int list) =
sum(l) + mul(l) + length(l)
```

Order of sum() and mul(), etc does not matter – they do not modify *l*

© Spinnaker Labs, Inc.

Functional Updates Do Not Modify Structures

```
fun append(x, lst) =
let lst' = reverse lst in
reverse ( x :: lst' )
```

The append() function above reverses a list, adds a new element to the front, and returns all of that, reversed, which appends an item.

But it *never modifies lst*!

© Spinnaker Labs, Inc.

Functions Can Be Used As Arguments

```
fun DoDouble(f, x) = f (f x)
```

It does not matter what f does to its argument; DoDouble() will do it twice.

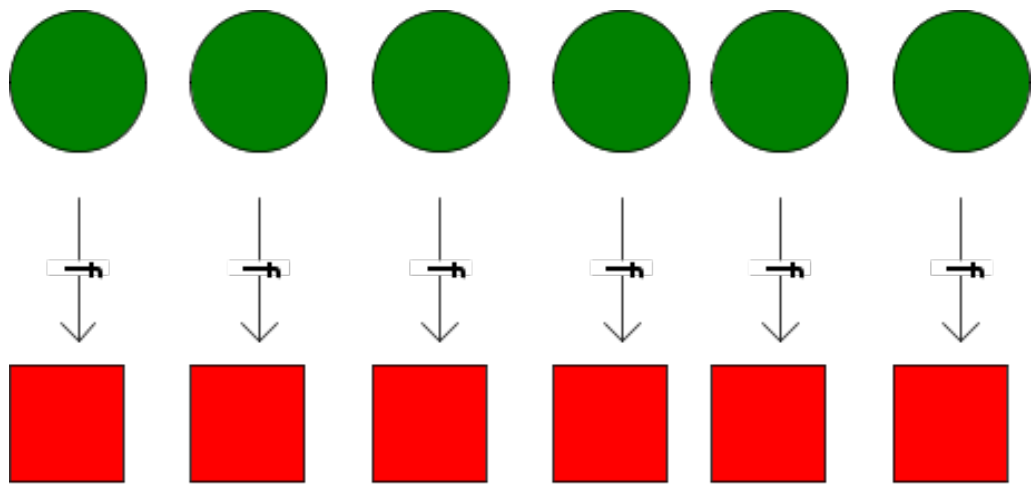
What is the type of this function?

© Spinnaker Labs, Inc.

Map

map f lst: (’a->’b) -> (’a list) -> (’b list)

Creates a new list by applying f to each element of the input list; returns output in order.

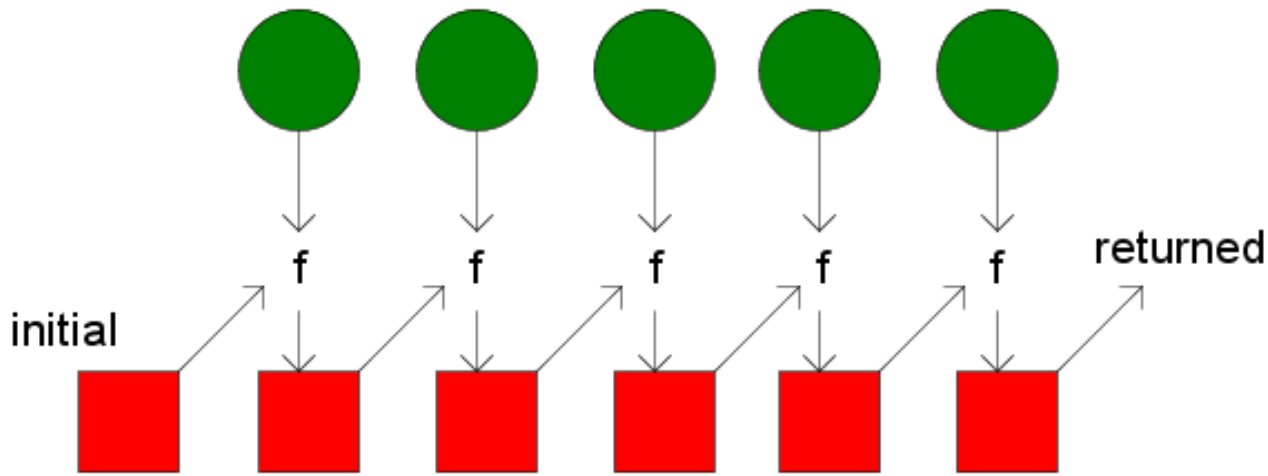


© Spinnaker Labs, Inc.

Fold

fold f x_0 lst: ('a*'b->'b)->'b->('a list)->'b

Moves across a list, applying *f* to each element plus an *accumulator*. f returns the next accumulator value, which is combined with the next element of the list



© Spinnaker Labs, Inc.

fold left vs. fold right

- Order of list elements can be significant
- Fold left moves left-to-right across the list
- Fold right moves from right-to-left

SML Implementation:

```
fun foldl f a [] = a
| foldl f a (x::xs) = foldl f (f(x, a)) xs

fun foldr f a [] = a
| foldr f a (x::xs) = f(x, (foldr f a xs))
```

© Spinnaker Labs, Inc.

Example

```
fun foo(l: int list) =
sum(l) + mul(l) + length(l)
```

How can we implement this?

© Spinnaker Labs, Inc.

Example (Solved)

```
fun foo(l: int list) =
sum(l) + mul(l) + length(l)

fun sum(lst) = foldl (fn (x,a)=>x+a) 0 lst
fun mul(lst) = foldl (fn (x,a)=>x*a) 1 lst
fun length(lst) = foldl (fn (x,a)=>1+a) 0 lst
```

© Spinnaker Labs, Inc.

map Implementation

```
fun map f [] = []
| map f (x::xs) = (f x) :: (map f xs)
```

- This implementation moves left-to-right across the list, mapping elements one at a time
- … But does it need to?

© Spinnaker Labs, Inc.

Implicit Parallelism In map

- In a purely functional setting, elements of a list being computed by map cannot see the effects of the computations on other elements
- If order of application of *f* to elements in list is *commutative*, we can reorder or parallelize execution
- This is the “secret” that MapReduce exploits

© Spinnaker Labs, Inc.

MapReduce

© Spinnaker Labs, Inc.

Motivation: Large Scale Data Processing

- Want to process lots of data (> 1 TB)
- Want to parallelize across hundreds/thousands of CPUs
- … Want to make this easy

© Spinnaker Labs, Inc.

MapReduce

- Automatic parallelization & distribution
- Fault-tolerant
- Provides status and monitoring tools
- Clean abstraction for programmers

© Spinnaker Labs, Inc.

Programming Model

- Borrows from functional programming
- Users implement interface of two functions:

- `map (in_key, in_value) -> (out_key, intermediate_value) list`

- `reduce (out_key, intermediate_value list) -> out_value list`

© Spinnaker Labs, Inc.

map

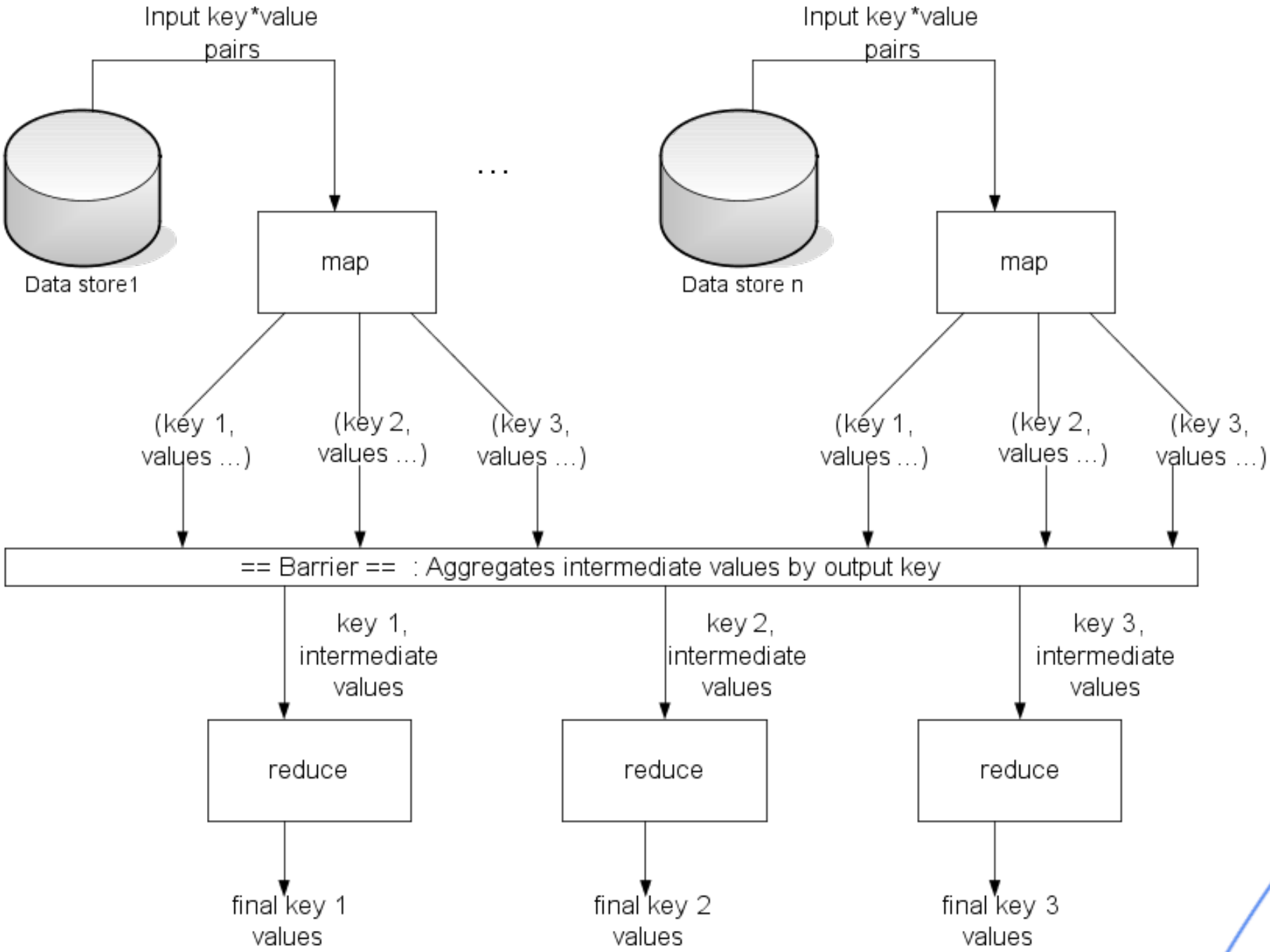
- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).
- map() produces one or more *intermediate* values along with an output key from the input.

© Spinnaker Labs, Inc.

reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)

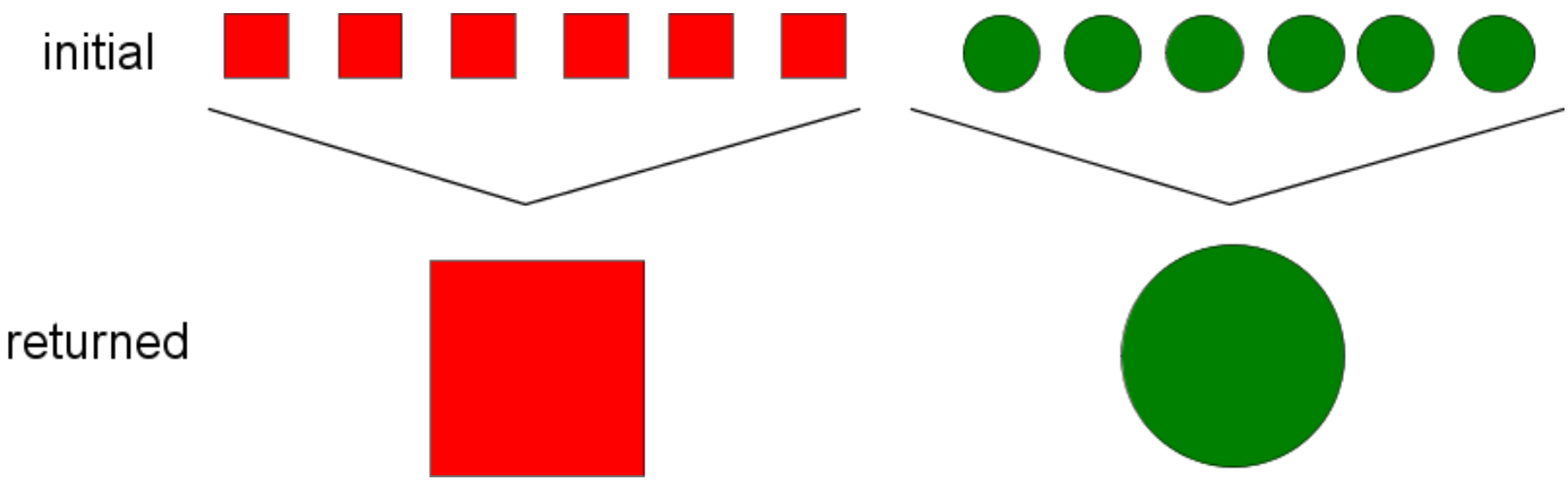
© Spinnaker Labs, Inc.



© Spinnaker Labs, Inc.

reduce

```
reduce (out_key, intermediate_value list) ->
out_value list
```



© Spinnaker Labs, Inc.

Parallelism

- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can't start until map phase is completely finished.

© Spinnaker Labs, Inc.

Example: Count word occurrences

```
map(String input_key, String input_value):
// input_key: document name
// input_value: document contents
for each word w in input_value:
EmitIntermediate(w, "1");


reduce(String output_key, Iterator
intermediate_values):
// output_key: a word
// output_values: a list of counts
int result = 0;
for each v in intermediate_values:
result += ParseInt(v);
```

Example vs. Actual Source Code

- Example is written in pseudo-code
- Actual implementation is in C++, using a MapReduce library
- Bindings for Python and Java exist via interfaces
- True code is somewhat more involved (defines how the input key/values are divided up and accessed, etc.)

© Spinnaker Labs, Inc.

Locality

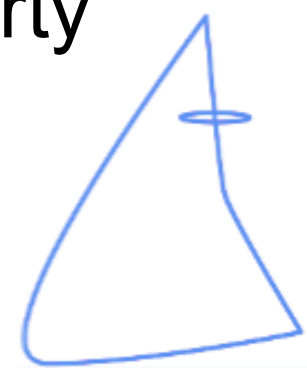
- Master program divvies up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

© Spinnaker Labs, Inc.

Fault Tolerance

- Master detects worker failures
  - Re-executes completed & in-progress map() tasks
  - Re-executes in-progress reduce() tasks
- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
  - Effect: Can work around bugs in third-party libraries!

© Spinnaker Labs, Inc.

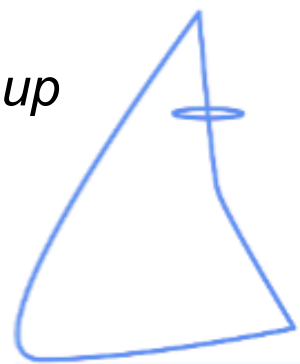


Optimizations

- No reduce can start until map is complete:
  - A single slow disk controller can rate-limit the whole process
- Master redundantly executes “slow-moving” map tasks; uses results of first copy to finish

Why is it safe to redundantly execute map tasks? Wouldn’t this mess up the total computation?

© Spinnaker Labs, Inc.



Optimizations

- "Combiner" functions can run on same machine as a mapper
- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth

Under what conditions is it sound to use a combiner?

© Spinnaker Labs, Inc.

The Example Again

```
map(String input_key, String input_value):
// input_key: document name
// input_value: document contents
for each word w in input_value:
EmitIntermediate(w, "1");


reduce(String output_key, Iterator
intermediate_values):
// output_key: a word
// output_values: a list of counts
int result = 0;
for each v in intermediate_values:
result += ParseInt(v);
```

MapReduce Conclusions

- MapReduce has proven to be a useful abstraction
- Greatly simplifies large-scale computations at Google
- Functional programming paradigm can be applied to large-scale applications
- Fun to use: focus on problem, let library deal w/ messy details

© Spinnaker Labs, Inc.

Part 2: Algorithms

© Spinnaker Labs, Inc.

Algorithms for MapReduce

- Sorting
- Searching
- Indexing
- Classification
- TF-IDF
- Breadth-First Search / SSSP
- PageRank
- Clustering

© Spinnaker Labs, Inc.

MapReduce Jobs

- Tend to be very short, code-wise
  - IdentityReducer is very common
- "Utility" jobs can be composed
- Represent a *data flow*, more so than a procedure

© Spinnaker Labs, Inc.

Sort: Inputs

- A set of files, one value per line.
- Mapper key is file name, line number
- Mapper value is the contents of the line

© Spinnaker Labs, Inc.

Sort Algorithm

- Takes advantage of reducer properties: (key, value) pairs are processed in order by key; reducers are themselves ordered

- Mapper: Identity function for value

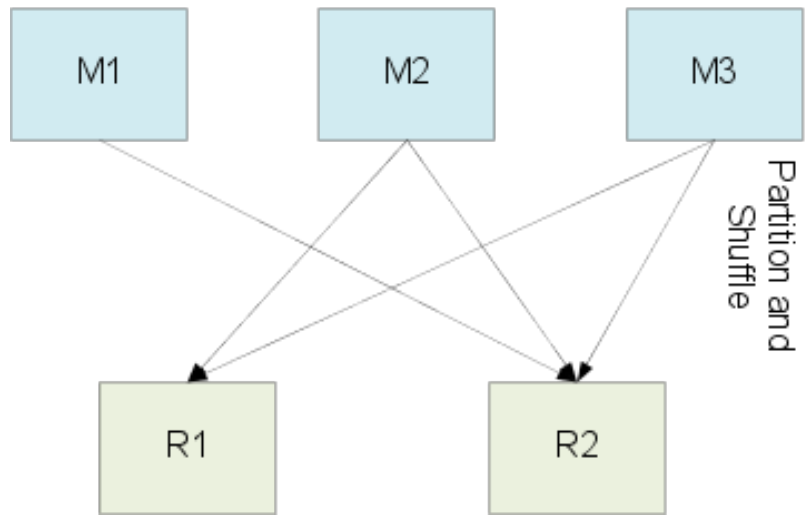
  (k, v) • (v, _)

- Reducer: Identity function (k', _) -> (k', "")

© Spinnaker Labs, Inc.

Sort: The Trick

- (key, value) pairs from mappers are sent to a particular reducer based on hash(key)
- Must pick the hash function for your data such that $k_1 < k_2 \Rightarrow hash(k_1) < hash(k_2)$



© Spinnaker Labs, Inc.

Final Thoughts on Sort

- Used as a test of Hadoop's raw speed
- Essentially "IO drag race"
- Highlights utility of GFS

© Spinnaker Labs, Inc.

Search: Inputs

- A set of files containing lines of text
- A search pattern to find

- Mapper key is file name, line number
- Mapper value is the contents of the line
- Search pattern sent as special parameter

© Spinnaker Labs, Inc.

Search Algorithm

- Mapper:
  - Given (filename, some text) and “pattern”, if “text” matches “pattern” output (filename, _)
- Reducer:
  - Identity function

© Spinnaker Labs, Inc.

Search: An Optimization

- Once a file is found to be interesting, we only need to mark it that way once
- Use *Combiner* function to fold redundant (filename, _) pairs into a single one
  - Reduces network I/O

© Spinnaker Labs, Inc.

Indexing: Inputs

- A set of files containing lines of text

- Mapper key is file name, line number
- Mapper value is the contents of the line

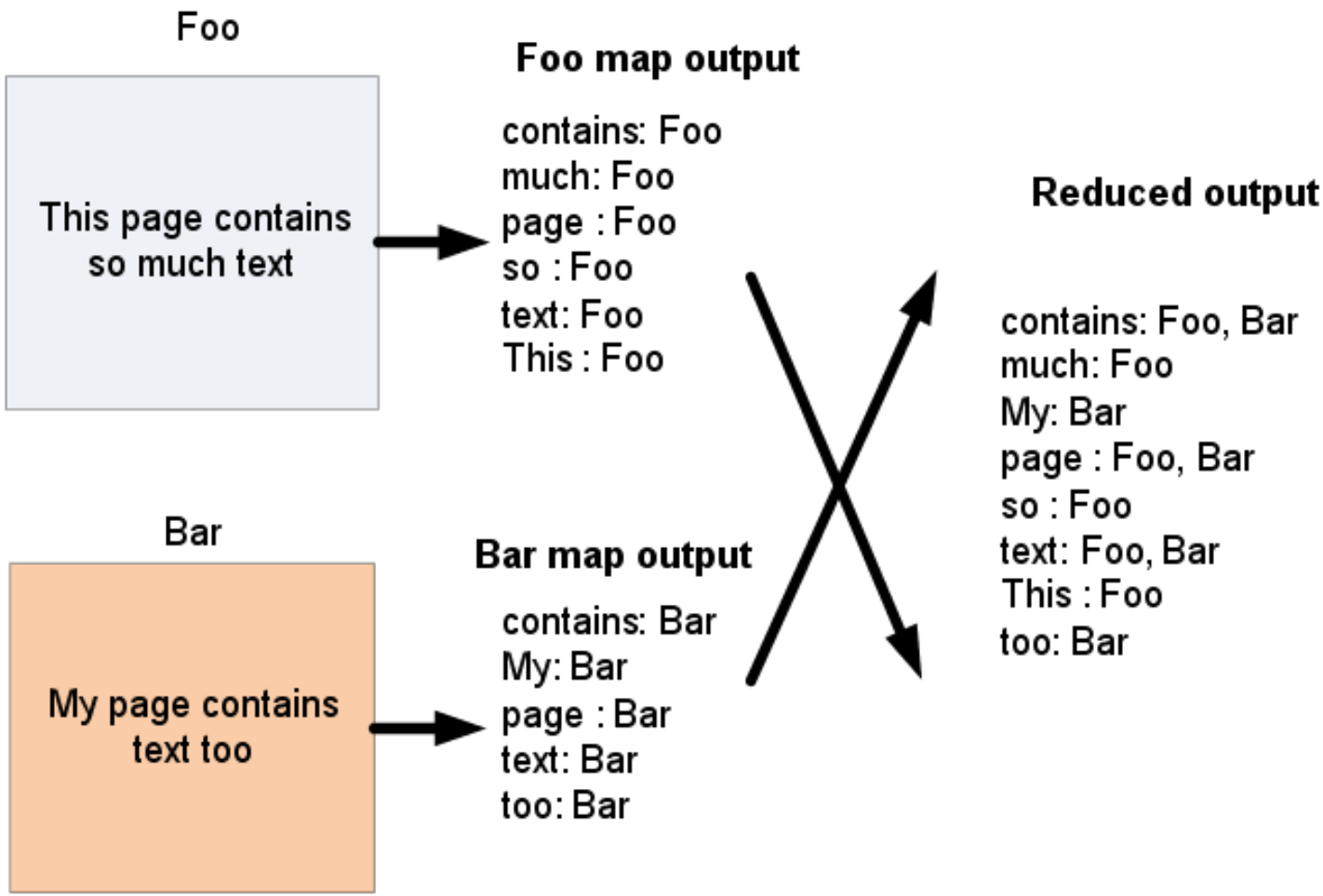
© Spinnaker Labs, Inc.

Inverted Index Algorithm

- Mapper: For each word in (file, words), map to (word, file)
- Reducer: Identity function

© Spinnaker Labs, Inc.

Inverted Index: Data flow



© Spinnaker Labs, Inc.

An Aside: Word Count

- Word count was described in module I
- Mapper for Word Count is (word, 1) for each word in input line
  - Strikingly similar to inverted index
  - Common theme: reuse/modify existing mappers

© Spinnaker Labs, Inc.

Bayesian Classification

- Files containing classification instances are sent to mappers
- Map (filename, instance) • (instance, class)
- Identity Reducer

© Spinnaker Labs, Inc.

Bayesian Classification

- Existing toolsets exist to perform Bayes classification on instance
  - E.g., WEKA, already in Java!
- Another example of discarding input key

© Spinnaker Labs, Inc.

TF-IDF

- Term Frequency – Inverse Document Frequency
  - Relevant to text processing
  - Common web analysis algorithm

© Spinnaker Labs, Inc.

The Algorithm, Formally

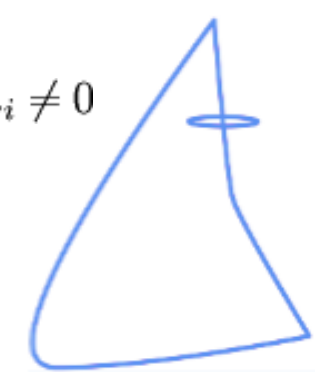
$$\mathrm{tf_i} = \frac{n_i}{\sum_k n_k}$$

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

$$\mathrm{tfidf} = \mathrm{tf} \cdot \mathrm{idf}$$

- $|D|$: total number of documents in the corpus
- $|\{d : t_i \in d\}|$: number of documents where the term t_i appears (that is $n_i \neq 0$).

© Spinnaker Labs, Inc.



Information We Need

- Number of times term X appears in a given document
- Number of terms in each document
- Number of documents X appears in
- Total number of documents

© Spinnaker Labs, Inc.

Job 1: Word Frequency in Doc

- Mapper
  - Input: (docname, contents)
  - Output: ((word, docname), 1)
- Reducer
  - Sums counts for word in document
  - Outputs ((word, docname), *n*)
- Combiner is same as Reducer

© Spinnaker Labs, Inc.

Job 2: Word Counts For Docs

- Mapper
  - Input: ((word, docname), *n*)
  - Output: (docname, (word, *n)*)
- Reducer
  - Sums frequency of individual *n*'s in same doc
  - Feeds original data through
  - Outputs ((word, docname), (*n*, *N)*)

© Spinnaker Labs, Inc.

Job 3: Word Frequency In Corpus

- Mapper
  - Input: ((word, docname), (*n, N)*)
  - Output: (word, (docname, *n*, *N*, 1))
- Reducer
  - Sums counts for word in corpus
  - Outputs ((word, docname), (*n*, *N*, *m*))

© Spinnaker Labs, Inc.

Job 4: Calculate TF-IDF

- Mapper
  - Input: ((word, docname), (n, N, m))
  - Assume D is known (or, easy MR to find it)
  - Output ((word, docname), TF*IDF)
- Reducer
  - Just the identity function

© Spinnaker Labs, Inc.

Final Thoughts on TF-IDF

- Several small jobs add up to full algorithm
- Lots of code reuse possible
  - Stock classes exist for aggregation, identity
- Jobs 3 and 4 can really be done at once in same reducer, saving a write/read cycle

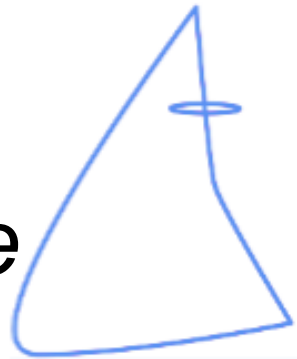
© Spinnaker Labs, Inc.

BFS: Motivating Concepts

- Performing computation on a graph data structure requires processing at each node
- Each node contains node-specific data as well as links (edges) to other nodes
- Computation must traverse the graph and perform the computation step

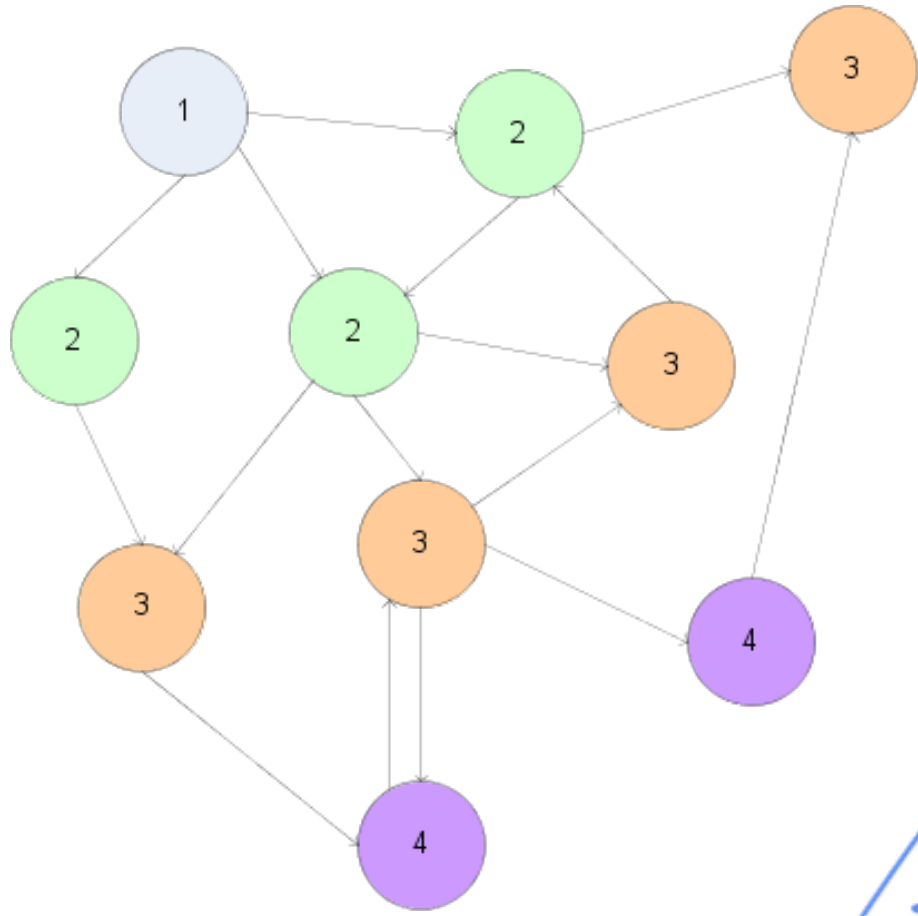
- *How do we traverse a graph in MapReduce? How do we represent the graph for this?*

© Spinnaker Labs, Inc.



Breadth-First Search

- Breadth-First Search is an *iterated* algorithm over graphs
- Frontier advances from origin by one level with each pass



© Spinnaker Labs, Inc.

Breadth-First Search & MapReduce

- Problem: This doesn't “fit” into MapReduce
- Solution: Iterated passes through MapReduce – map some nodes, result includes additional nodes which are fed into successive MapReduce passes

© Spinnaker Labs, Inc.

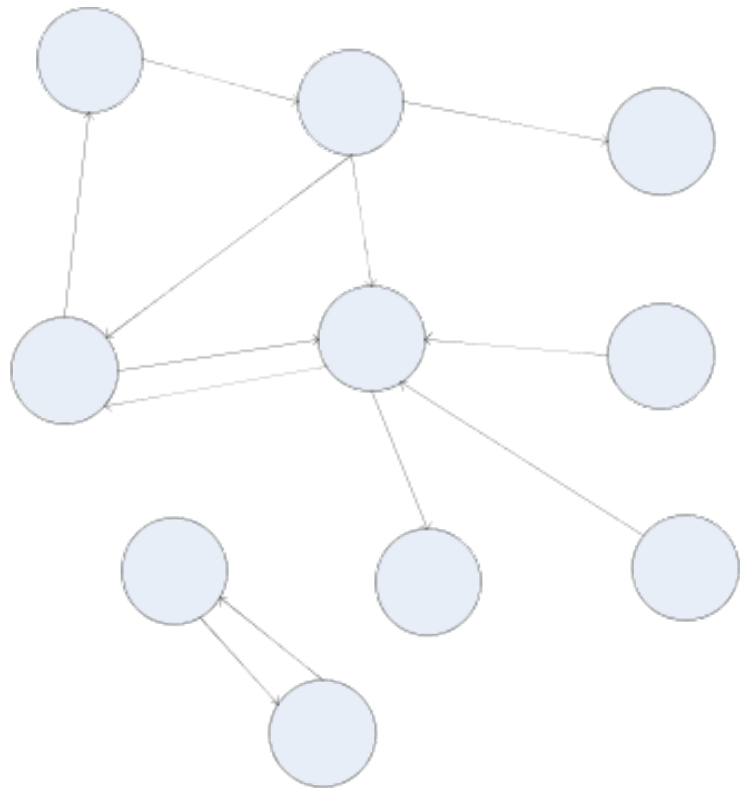
Breadth-First Search & MapReduce

- Problem: Sending the entire graph to a map task (or hundreds/thousands of map tasks) involves an enormous amount of memory
- Solution: Carefully consider how we represent graphs

© Spinnaker Labs, Inc.

Graph Representations

- The most straightforward representation of graphs uses references from each node to its neighbors



© Spinnaker Labs, Inc.

Direct References

- Structure is inherent to object
- Iteration requires linked list “threaded through” graph
- Requires common view of shared memory (synchronization!)
- Not easily

```
class GraphNode
{
Object data;
Vector<GraphNode>
out_edges;
GraphNode
iter_next;
}
```

© Spinnaker Labs, Inc.

Adjacency Matrices

- Another classic graph representation. M[i][j]= '1' implies a link from node *i to j.*
- Naturally encapsulates iteration over nodes

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	0
4	1	0	1	0

Adjacency Matrices: Sparse Representation

- Adjacency matrix for most large graphs (e. g., the web) will be overwhelmingly full of zeros.
- Each row of the graph is absurdly long
- Sparse matrices only include non-zero elements

© Spinnaker Labs, Inc.

Sparse Matrix Representation

1: (3, 1), (18, 1), (200, 1)

2: (6, 1), (12, 1), (80, 1), (400, 1)

3: (1, 1), (14, 1)

…

© Spinnaker Labs, Inc.

Sparse Matrix Representation

1: 3, 18, 200

2: 6, 12, 80, 400

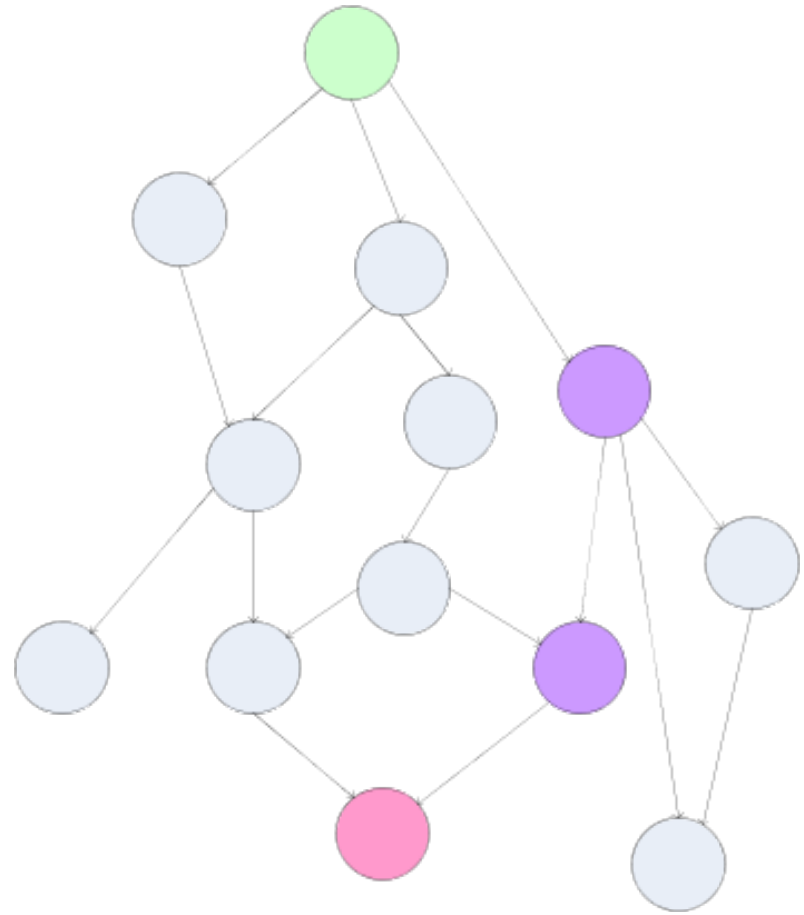
3: 1, 14

…

© Spinnaker Labs, Inc.

Finding the Shortest Path

- A common graph search application is finding the shortest path from a start node to one or more target nodes
- Commonly done on a single machine with *Dijkstra's Algorithm*
- Can we use BFS to find the shortest path via MapReduce?



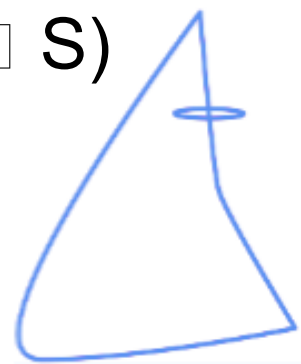
This is called the single-source shortest path problem. (a.k.a. SSSP)

Finding the Shortest Path: Intuition

- We can define the solution to this problem inductively:
  - DistanceTo(startNode) = 0
  - For all nodes *n* directly reachable from startNode, DistanceTo(n) = 1
  - For all nodes *n* reachable from some other set of nodes *S*,

    DistanceTo(n) = 1 + min(DistanceTo(m), m □ S)

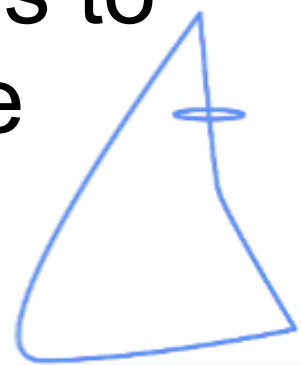
© Spinnaker Labs, Inc.



From Intuition to Algorithm

- A map task receives a node *n as* a key, and *(D, points-to)* as its value
  - *D* is the distance to the node from the start
  - *points-to* is a list of nodes reachable from *n*
  - p □ points-to, emit (p, D+1)

- Reduce task gathers possible distances to a given *p* and selects the minimum one

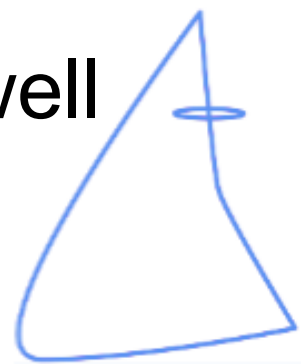


© Spinnaker Labs, Inc.

What This Gives Us

- This MapReduce task can advance the known frontier by one hop
- To perform the whole BFS, a non-MapReduce component then feeds the output of this step back into the MapReduce task for another iteration
  - Problem: Where'd the points-to list go?
  - Solution: Mapper emits (n, points-to) as well

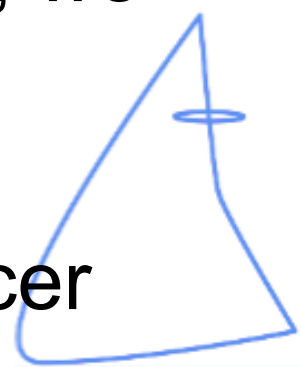
© Spinnaker Labs, Inc.



Blow-up and Termination

- This algorithm starts from one node
- Subsequent iterations include many more nodes of the graph as frontier advances
- Does this ever terminate?
  - Yes! Eventually, routes between nodes will stop being discovered and no better distances will be found. When distance is the same, we stop
  - Mapper should emit *(n, D)* to ensure that "current distance" is carried into the reducer

© Spinnaker Labs, Inc.



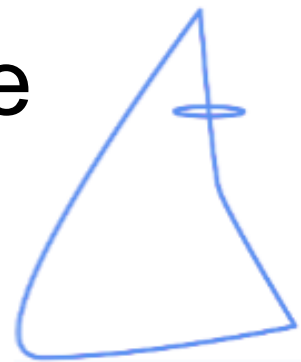
Adding weights

- Weighted-edge shortest path is more useful than cost==1 approach
- Simple change: points-to list in map task includes a weight 'w' for each pointed-to node
  - emit (p, D+w_p) instead of (p, D+1) for each node p
  - Works for positive-weighted graph

© Spinnaker Labs, Inc.

Comparison to Dijkstra

- Dijkstra's algorithm is more efficient because at any step it only pursues edges from the minimum-cost path inside the frontier
- MapReduce version explores all paths in parallel; not as efficient overall, but the architecture is more scalable
- Equivalent to Dijkstra for weight=1 case



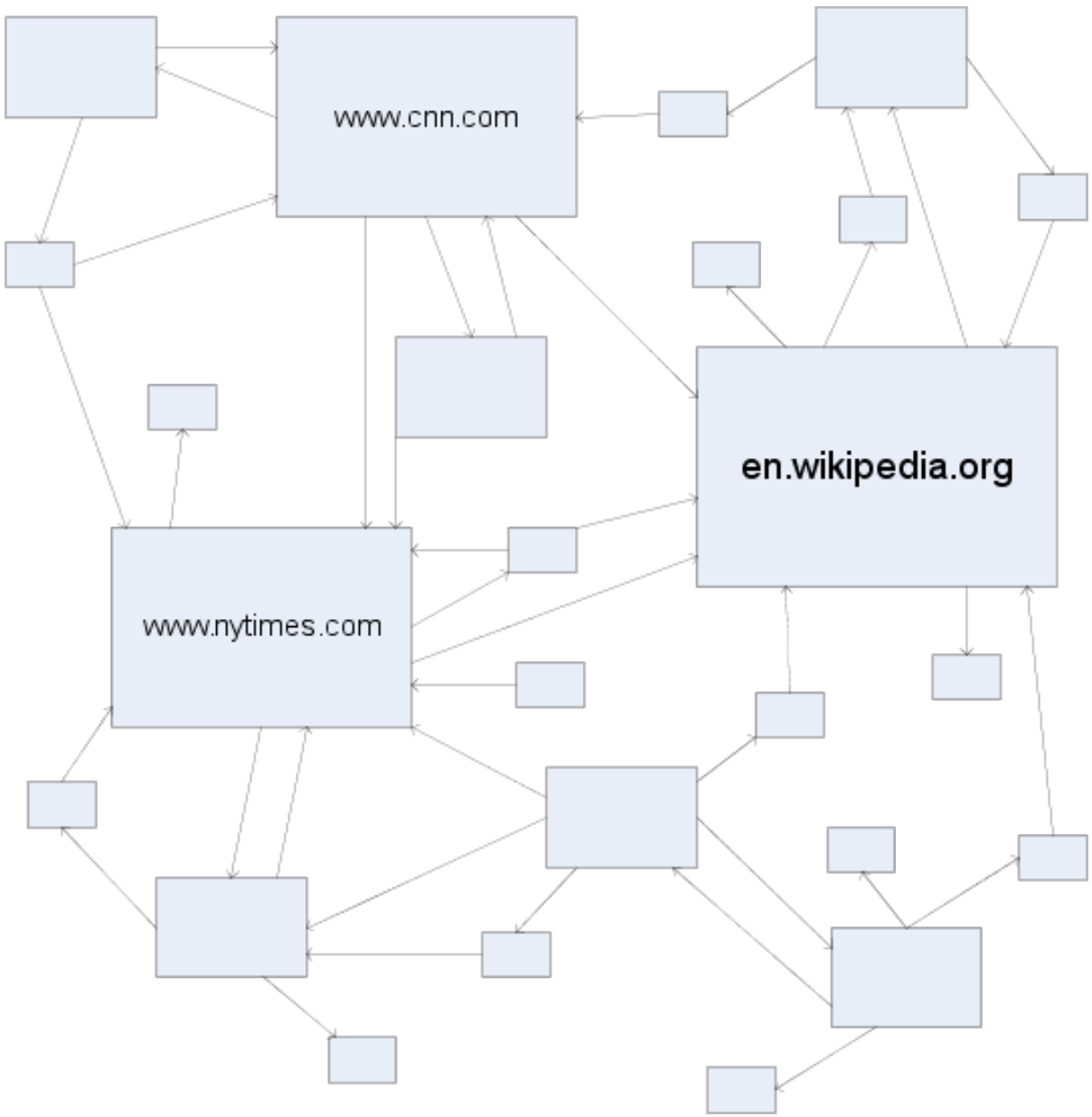
© Spinnaker Labs, Inc.

PageRank: Random Walks Over The Web

- If a user starts at a random web page and surfs by clicking links and randomly entering new URLs, what is the probability that s/he will arrive at a given page?
- The *PageRank* of a page captures this notion
  - More “popular” or “worthwhile” pages get a higher rank

© Spinnaker Labs, Inc.

PageRank: Visually



PageRank: Formula

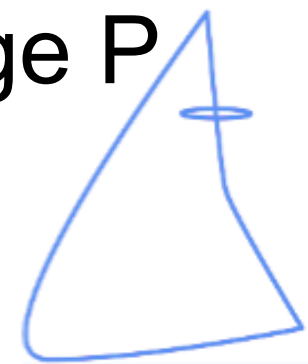
Given page A, and pages T_1 through T_n linking to A, PageRank is defined as:

$$PR(A) = (1-d) + d\ (PR(T_1)/C(T_1) + \ldots + PR(T_n)/C(T_n))$$

C(P) is the cardinality (out-degree) of page P

d is the damping (“random URL”) factor

© Spinnaker Labs, Inc.

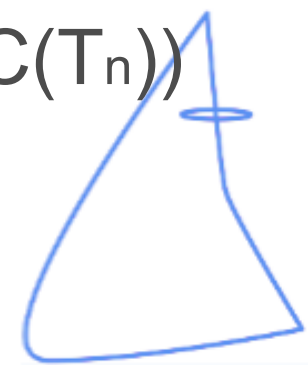


PageRank: Intuition

- Calculation is iterative: PR_{i+1} is based on PR_i
- Each page distributes its PR_i to all pages it links to. Linkees add up their awarded rank fragments to find their PR_{i+1}
- *d* is a tunable parameter (usually = 0.85) encapsulating the "random jump factor"

$$PR(A) = (1-d) + d\ (PR(T_1)/C(T_1) + \ldots + PR(T_n)/C(T_n))$$

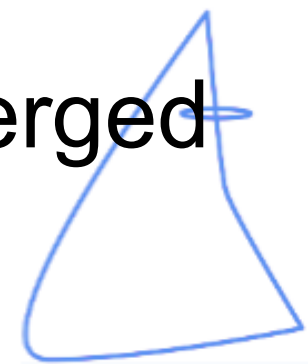
© Spinnaker Labs, Inc.



PageRank: First Implementation

- Create two tables 'current' and 'next' holding the PageRank for each page. Seed 'current' with initial PR values
- Iterate over all pages in the graph, distributing PR from 'current' into 'next' of linkees
- current := next; next := fresh_table();
- Go back to iteration step or end if converged

© Spinnaker Labs, Inc.



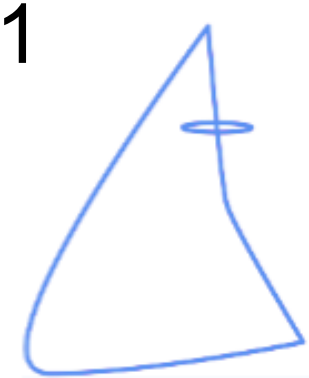
Distribution of the Algorithm

- Key insights allowing parallelization:
  - The 'next' table depends on 'current', but not on any other rows of 'next'
  - Individual rows of the adjacency matrix can be processed in parallel
  - Sparse matrix rows are relatively small

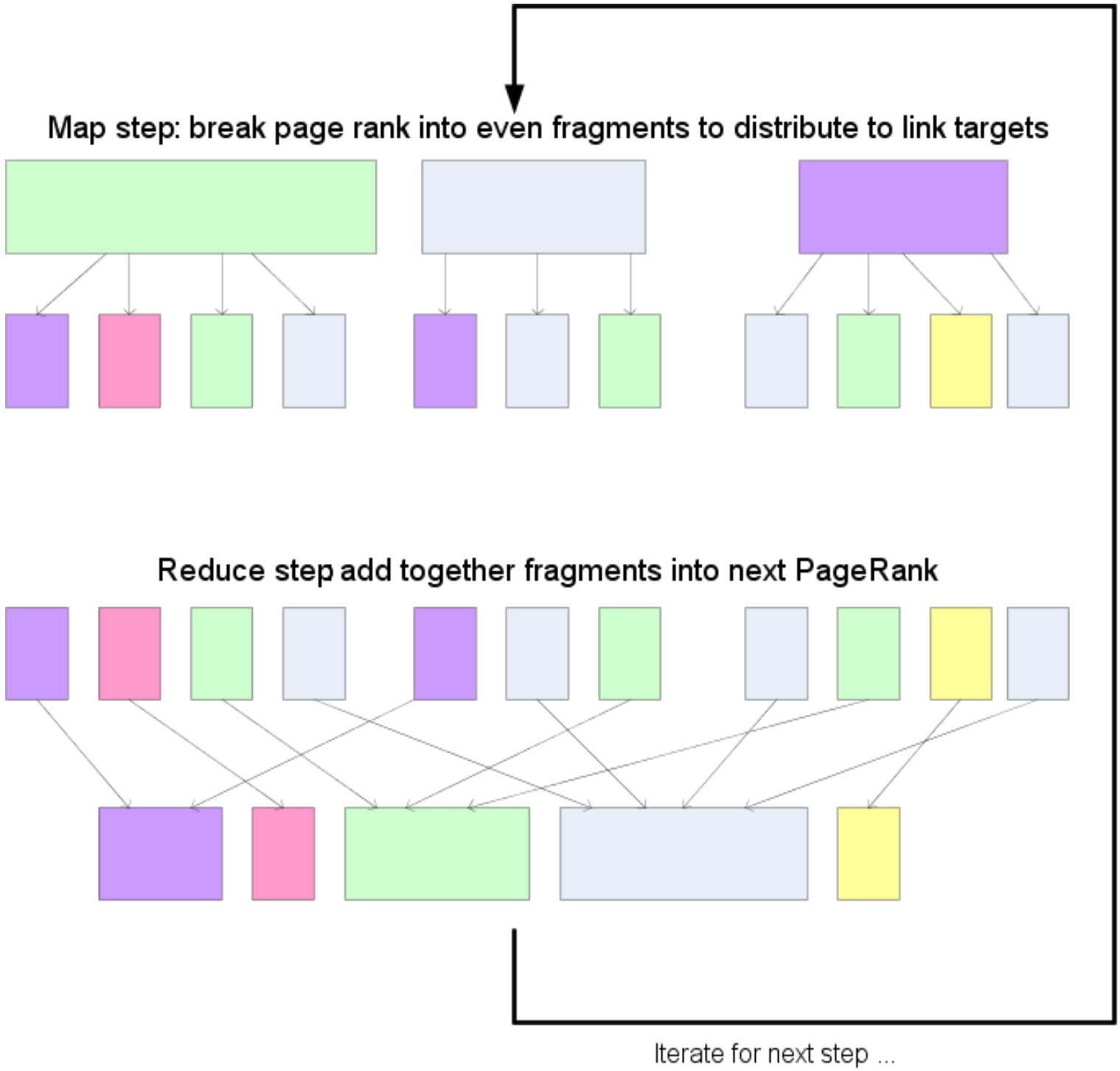
© Spinnaker Labs, Inc.

Distribution of the Algorithm

- Consequences of insights:
  - We can *map* each row of 'current' to a list of PageRank “fragments” to assign to linkees
  - These fragments can be *reduced* into a single PageRank value for a page by summing
  - Graph representation can be even more compact; since each element is simply 0 or 1, only transmit column numbers where it's 1



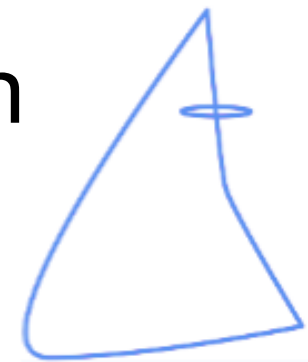
© Spinnaker Labs, Inc.


Map step: break page rank into even fragments to distribute to link targets
Reduce step add together fragments into next PageRank
Iterate for next step ...

Phase 1: Parse HTML

- Map task takes (URL, page content) pairs and maps them to (URL, (PR_{init}, list-of-urls))
  - PR_{init} is the "seed" PageRank for URL
  - list-of-urls contains all pages pointed to by URL
- Reduce task is just the identity function

© Spinnaker Labs, Inc.



Phase 2: PageRank Distribution

- Map task takes (URL, (cur_rank, url_list))
  - For each *u* in url_list, emit (*u*, cur_rank/|url_list|)
  - Emit (URL, url_list) to carry the points-to list along through iterations

$$PR(A) = (1-d) + d\ (PR(T_1)/C(T_1) + \ldots + PR(T_n)/C(T_n))$$

© Spinnaker Labs, Inc.

Phase 2: PageRank Distribution

- Reduce task gets (URL, url_list) and many (URL, *val*) values
  - Sum *val*s and fix up with *d*
  - Emit (URL, (new_rank, url_list))

$$PR(A) = (1-d) + d\ (PR(T_1)/C(T_1) + \ldots + PR(T_n)/C(T_n))$$

© Spinnaker Labs, Inc.

Finishing up...

- A non-parallelizable component determines whether convergence has been achieved (Fixed number of iterations? Comparison of key values?)
- If so, write out the PageRank lists - done!
- Otherwise, feed output of Phase 2 into another Phase 2 iteration

© Spinnaker Labs, Inc.

PageRank Conclusions

- MapReduce isn't the greatest at iterated computation, but still helps run the “heavy lifting”
- Key element in parallelization is independent PageRank computations in a given step
- Parallelization requires thinking about minimum data partitions to transmit (e.g., compact representations of graph rows)
  - Even the implementation shown today doesn't actually scale to the whole Internet; but it

Clustering

- What is clustering?

© Spinnaker Labs, Inc.

Google News

IPhone activation headaches still trouble users
Computerworld - **1 hour ago**
July 02, 2007 (Computerworld) -- It took Iain Gillott 47 hours to activate his iPhone after waiting in the Texas heat Friday afternoon to buy one.
Most iPhone users thrilled but a few are iRate Reuters
Apple iPhone Arrives in the US Techtree.com
Forbes - ZDNet - Ars Technica - Wired News
all 562 news articles »


Local6.com

McCain Considers Ways to Reshape Campaign
Washington Post - **35 minutes ago**
By Alec MacGillis Sen. John McCain's presidential campaign today announced widespread cutbacks and said it was considering whether to accept public campaign funds after another disappointing fundraising effort that has left the Arizona Republican with ...
McCain's Troubles Mount New York Times
McCain Campaign Struggling, Reduces Staff ABC News
CBS News - Reuters - Angus Reid Global Monitor - Sarasota Herald-Tribune
all 291 news articles »


Seattle Post Intelligencer

- They didn't pick all 3,400,217 related articles by hand…
- Or Amazon.com
- Or Netflix…

© Spinnaker Labs, Inc.

Other less glamorous things...

- Hospital Records
- Scientific Imaging
  - Related genes, related stars, related sequences
- Market Research
  - Segmenting markets, product positioning
- Social Network Analysis
- Data mining
- Image segmentation…

© Spinnaker Labs, Inc.

The Distance Measure

- How the similarity of two elements in a set is determined, e.g.
  - Euclidean Distance
  - Manhattan Distance
  - Inner Product Space
  - Maximum Norm
  - Or any metric you define over the space…

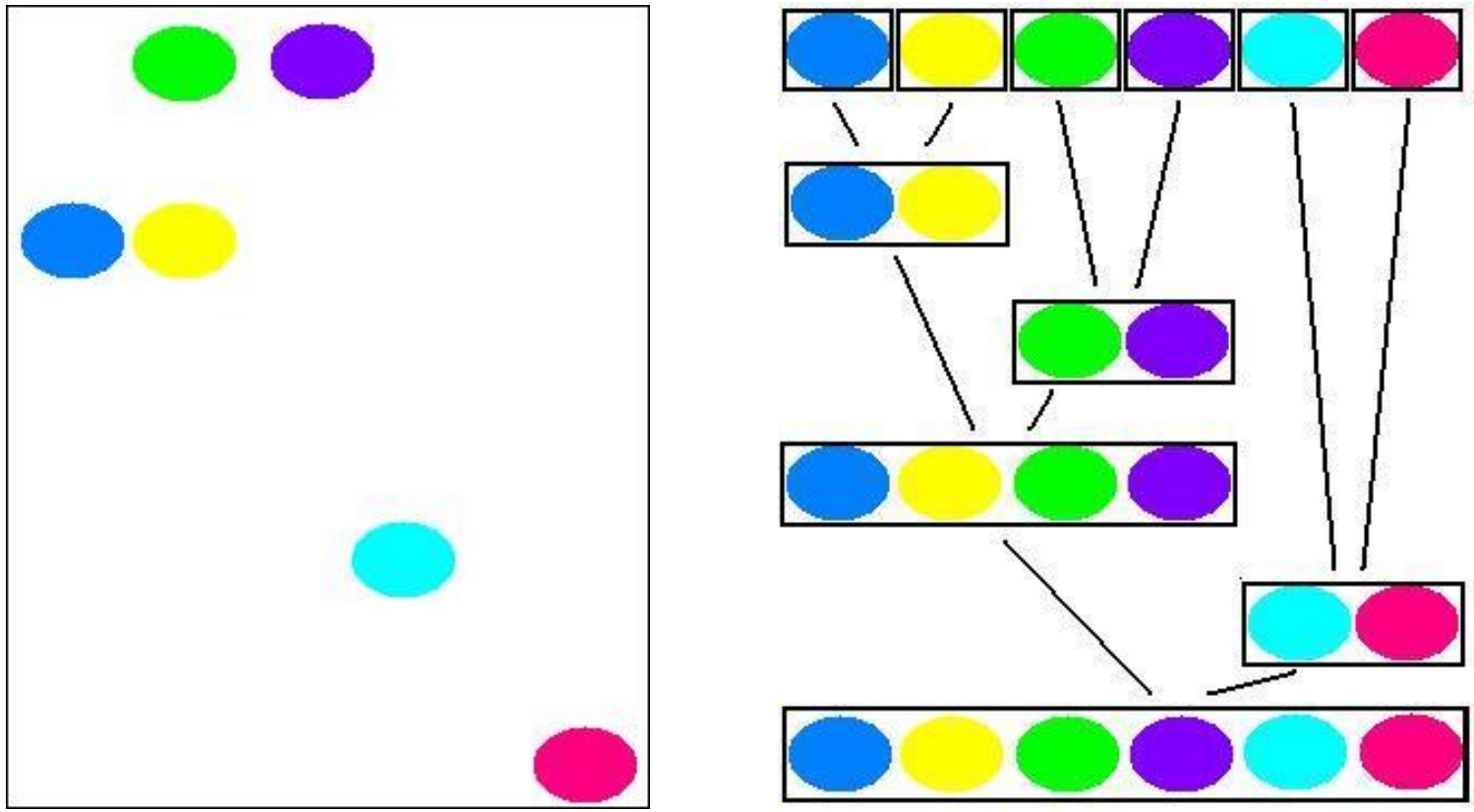
© Spinnaker Labs, Inc.

Types of Algorithms

- Hierarchical Clustering vs.
- Partitional Clustering

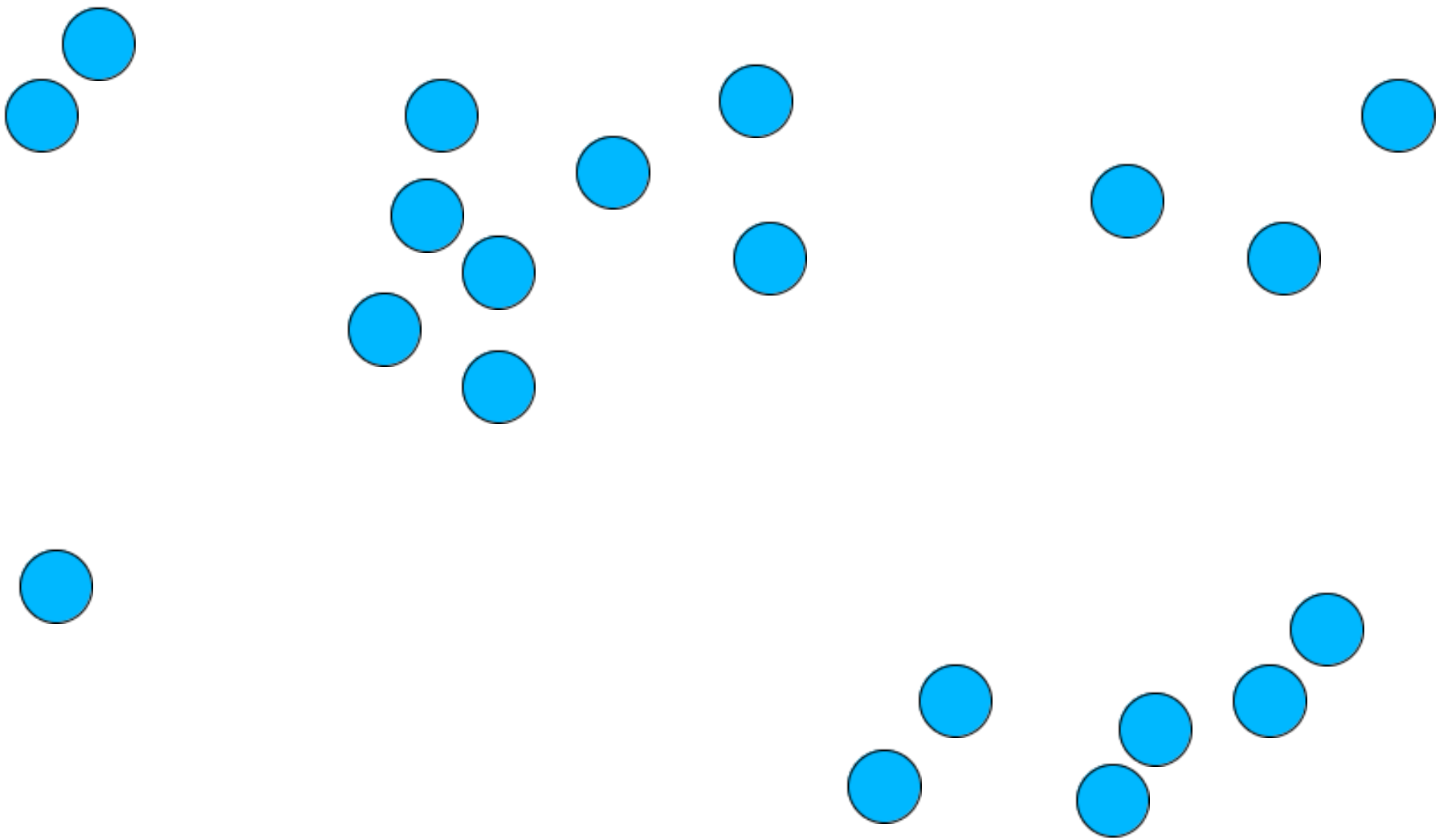
© Spinnaker Labs, Inc.

Hierarchical Clustering



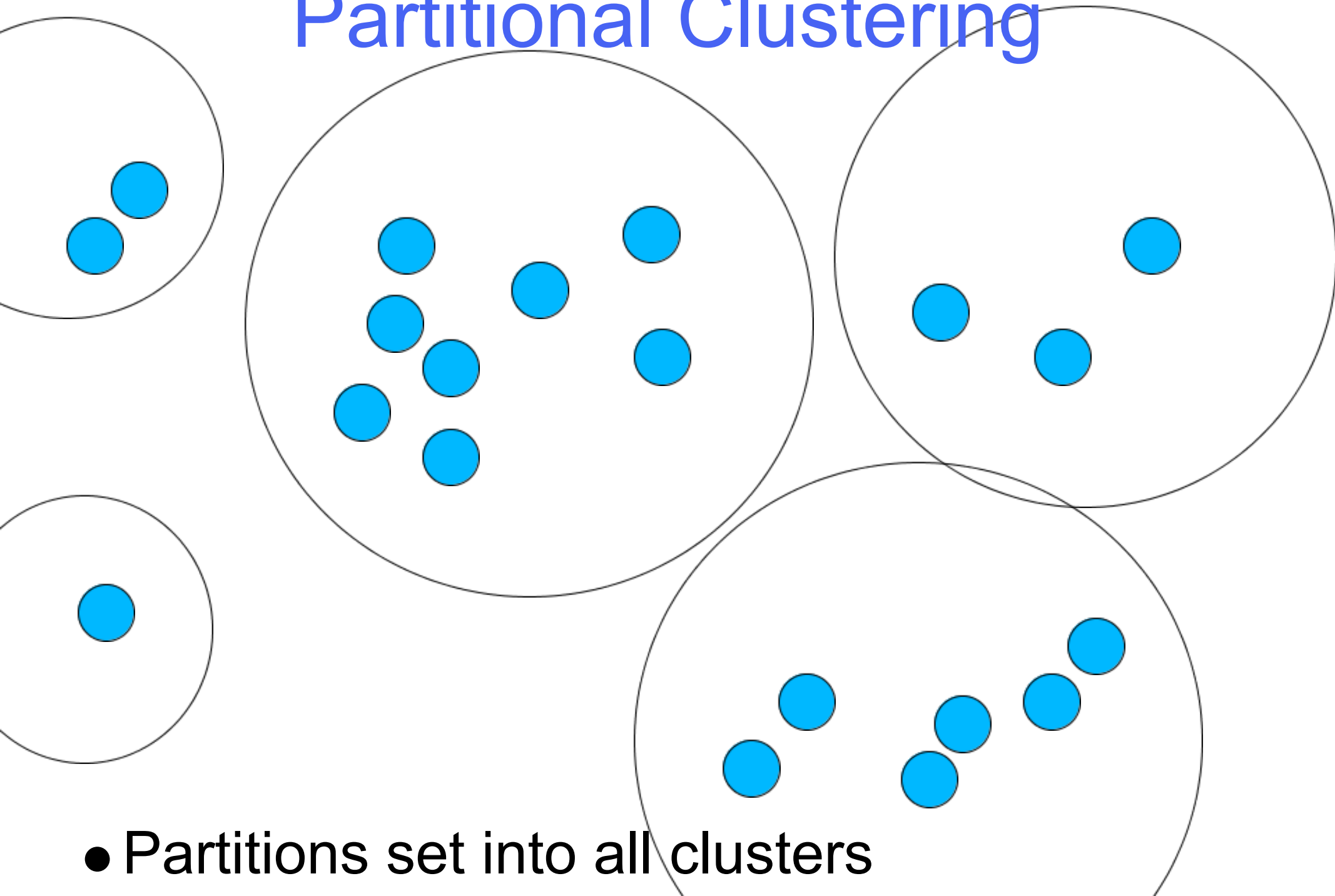
- Builds or breaks up a hierarchy of clusters.

Partitional Clustering



- Partitions set into all clusters

Partitional Clustering



- Partitions set into all clusters

K-Means Clustering

- Simple Partitional Clustering

- Choose the number of clusters, k
- Choose k points to be cluster centers
- Then…

© Spinnaker Labs, Inc.

K-Means Clustering

```
iterate {
Compute distance from all points to all k-
centers
Assign each point to the nearest k-center
Compute the average of all points assigned to
all specific k-centers
Replace the k-centers with the new averages
}
```

© Spinnaker Labs, Inc.

But!

- The complexity is pretty high:
  - k * n * O (distance metric) * num (iterations)
- Moreover, it can be necessary to send **tons** of data to each Mapper Node. Depending on your bandwidth and memory available, this could be impossible.

© Spinnaker Labs, Inc.

Furthermore

- There are three big ways a data set can be large:
  - There are a large number of elements in the set.
  - Each element can have many features.
  - There can be many clusters to discover
- Conclusion – Clustering can be huge, even when you distribute it.

© Spinnaker Labs, Inc.

Canopy Clustering

- Preliminary step to help parallelize computation.
- Clusters data into overlapping Canopies using super cheap distance metric.
- Efficient
- Accurate

© Spinnaker Labs, Inc.

Canopy Clustering

```
While there are unmarked points {
pick a point which is not strongly marked
call it a canopy center
mark all points within some threshold of
it as in it's canopy
strongly mark all points within some
stronger threshold
}
```

© Spinnaker Labs, Inc.

After the canopy clustering…

- Resume hierarchical or partitional clustering as usual.
- Treat objects in separate clusters as being at infinite distances.

© Spinnaker Labs, Inc.

MapReduce Implementation:

- Problem – Efficiently partition a large data set (say… movies with user ratings!) into a fixed number of clusters using Canopy Clustering, K-Means Clustering, and a Euclidean distance measure.

© Spinnaker Labs, Inc.

The Distance Metric

- The Canopy Metric ($)
- The K-Means Metric ($$$)

© Spinnaker Labs, Inc.

Steps!

- Get Data into a form you can use (MR)
- Picking Canopy Centers (MR)
- Assign Data Points to Canopies (MR)
- Pick K-Means Cluster Centers
- K-Means algorithm (MR)
  - Iterate!

© Spinnaker Labs, Inc.

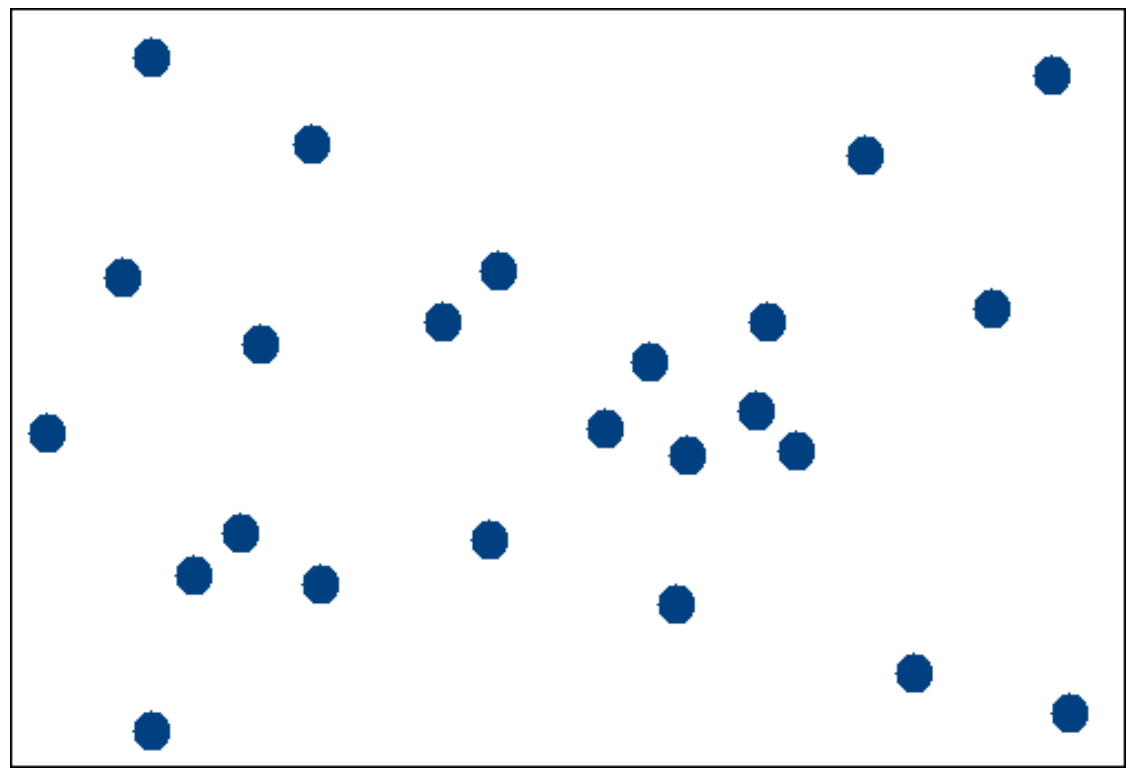
Data Massage

- This isn’t interesting, but it has to be done.

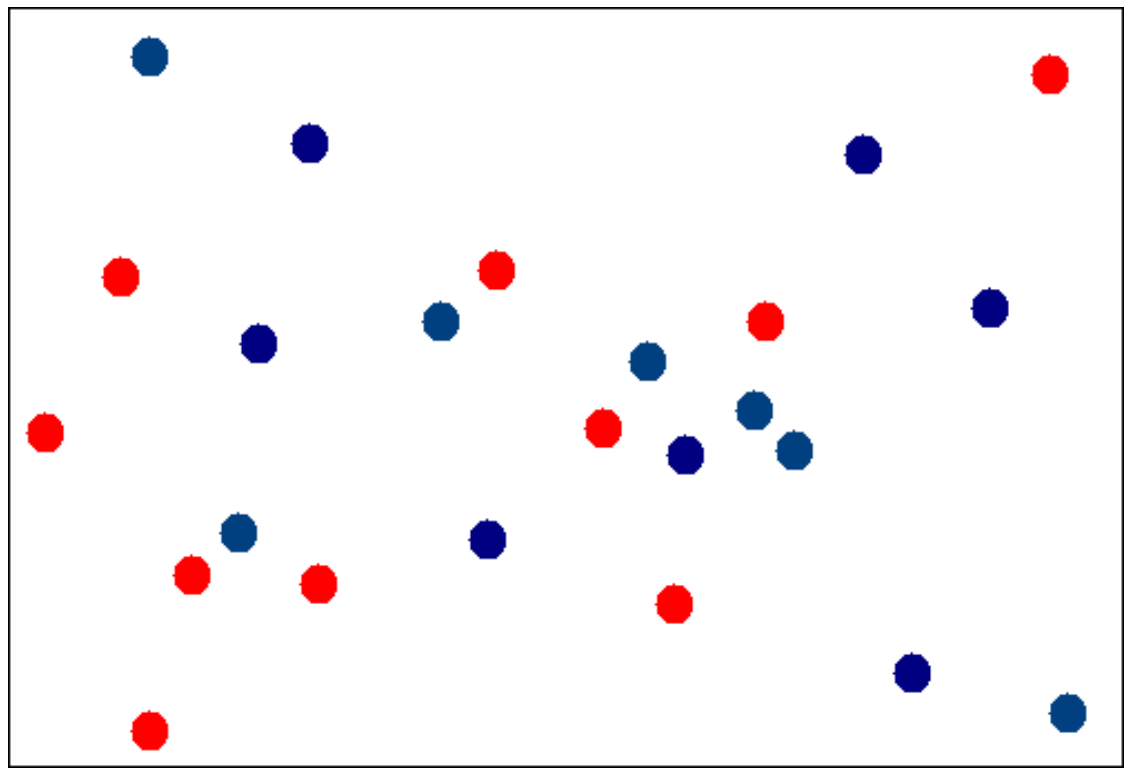
© Spinnaker Labs, Inc.

Selecting Canopy Centers

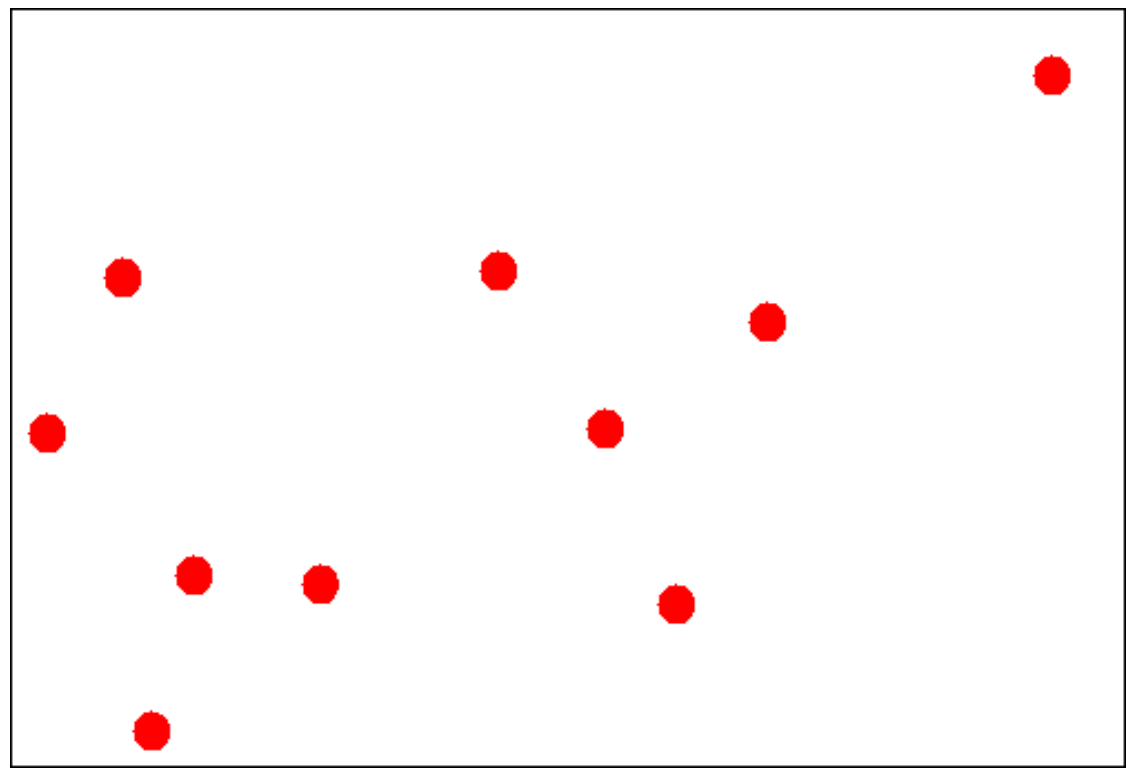
© Spinnaker Labs, Inc.



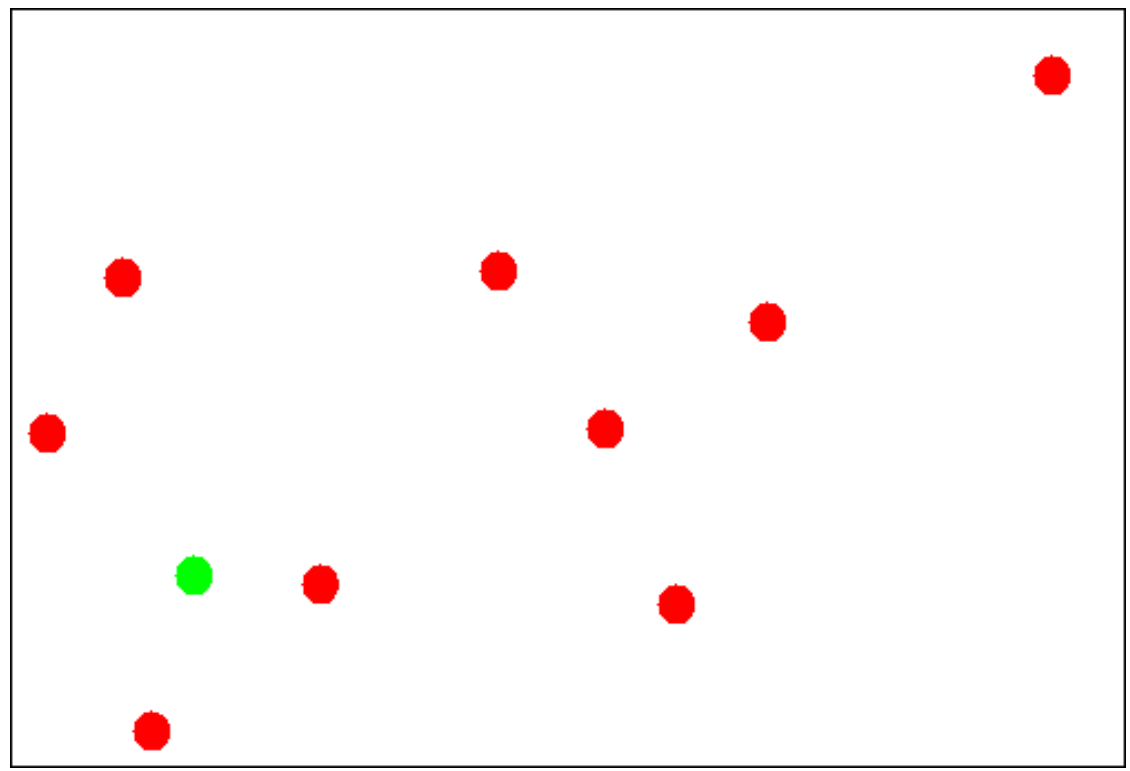
© Spinnaker Labs, Inc.



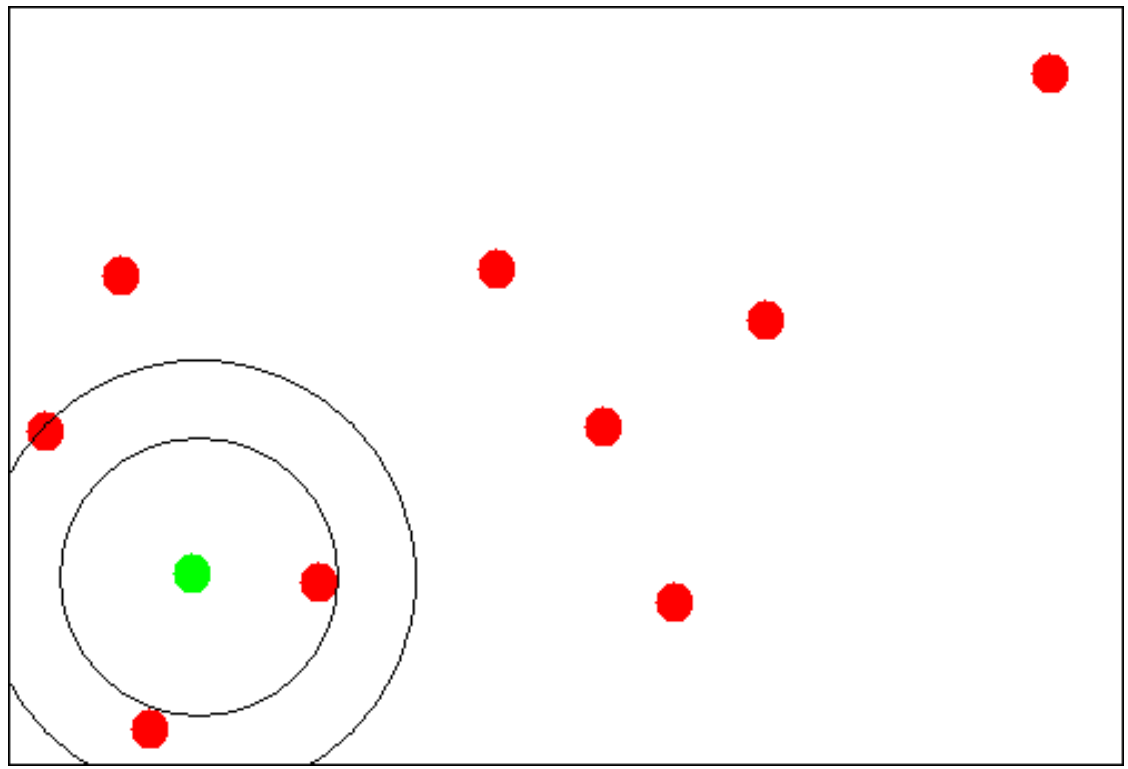
© Spinnaker Labs, Inc.



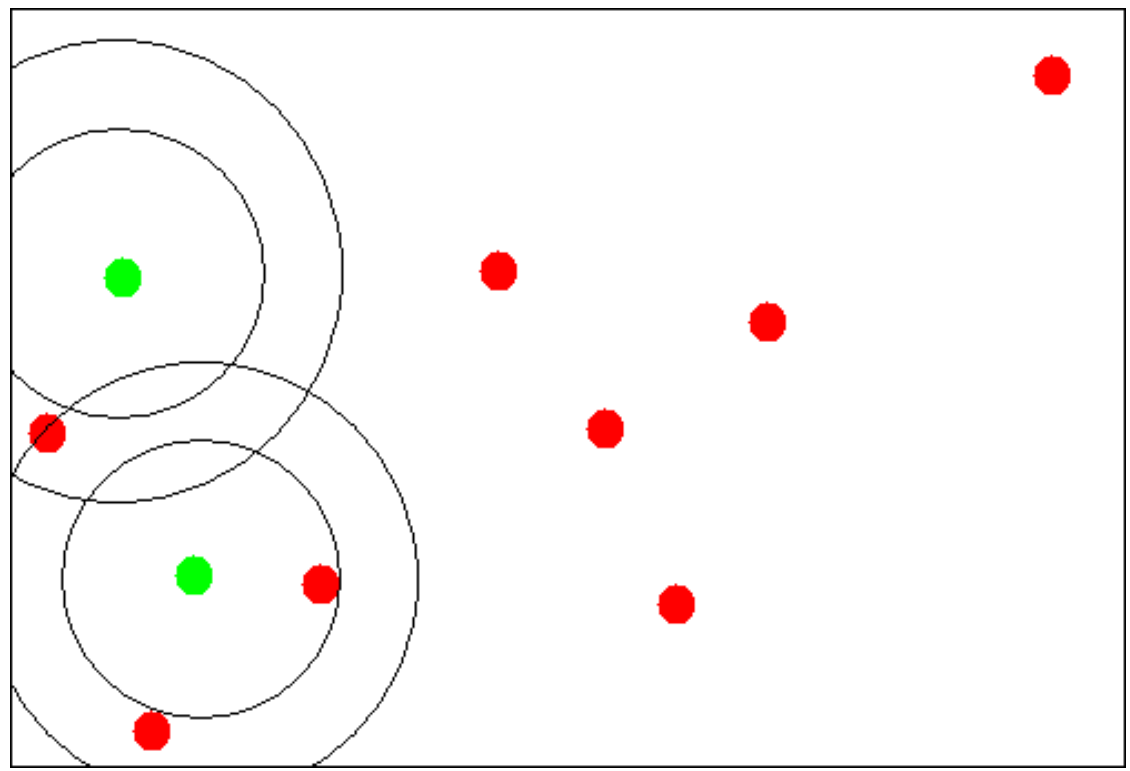
© Spinnaker Labs, Inc.



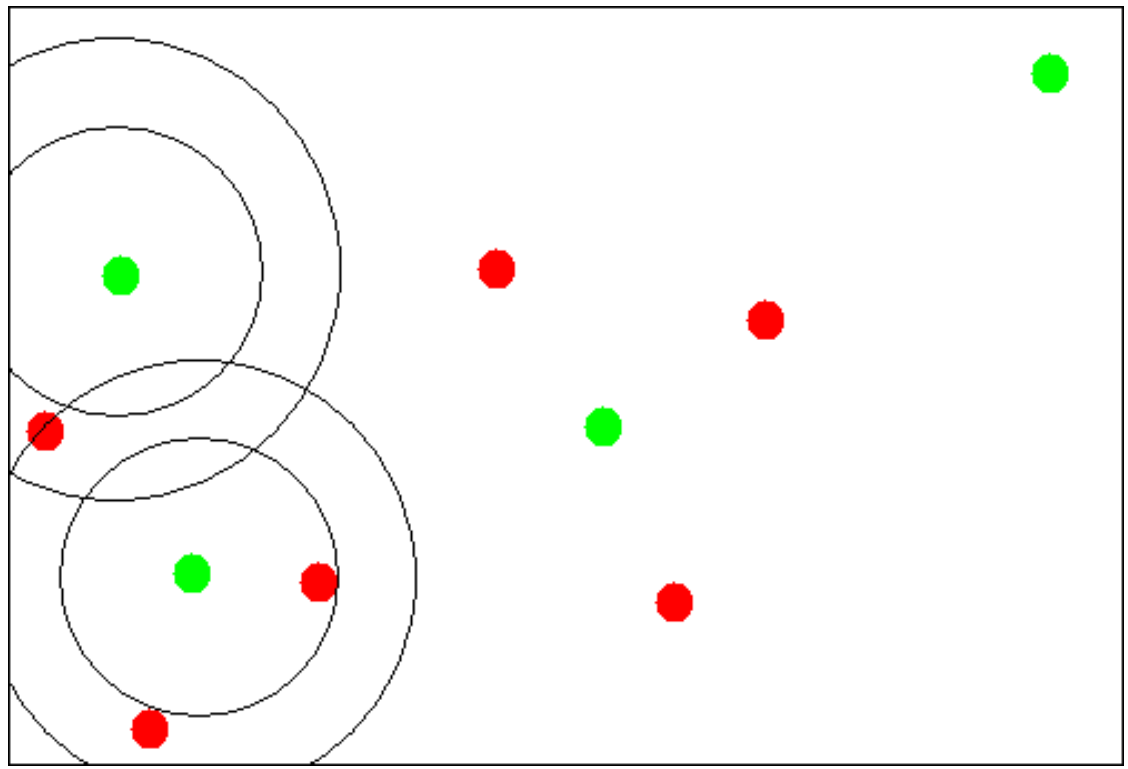
© Spinnaker Labs, Inc.



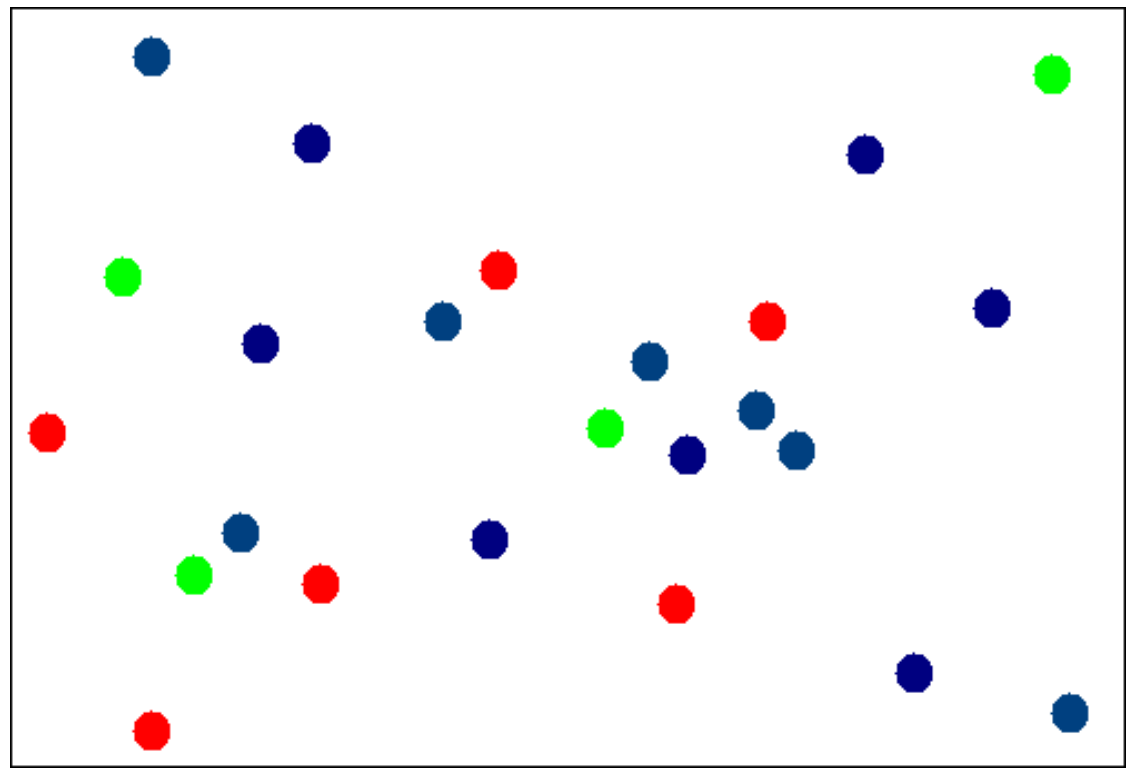
© Spinnaker Labs, Inc.



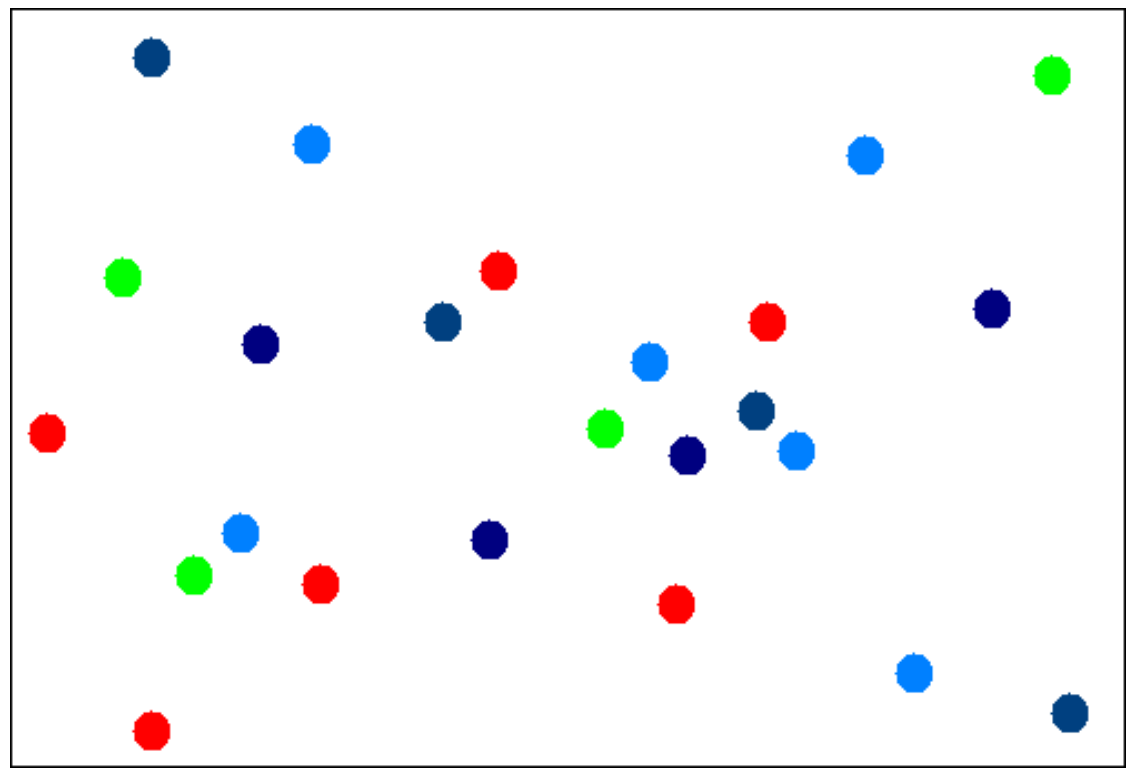
© Spinnaker Labs, Inc.



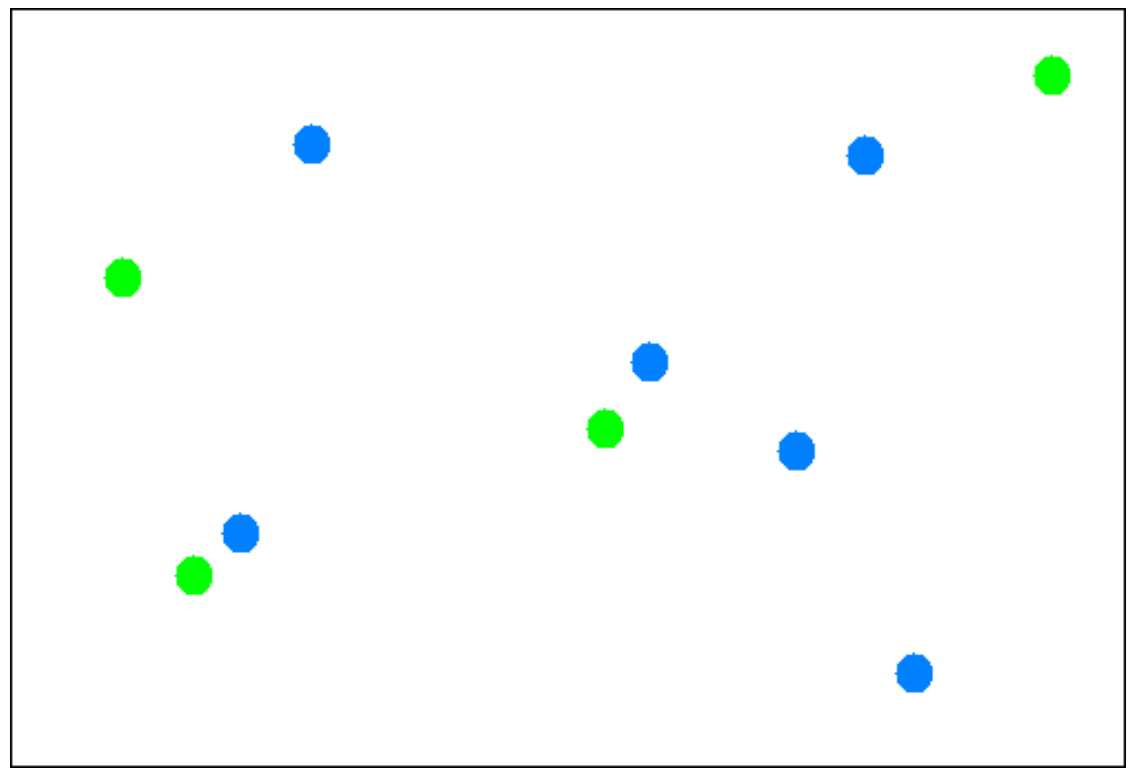
© Spinnaker Labs, Inc.



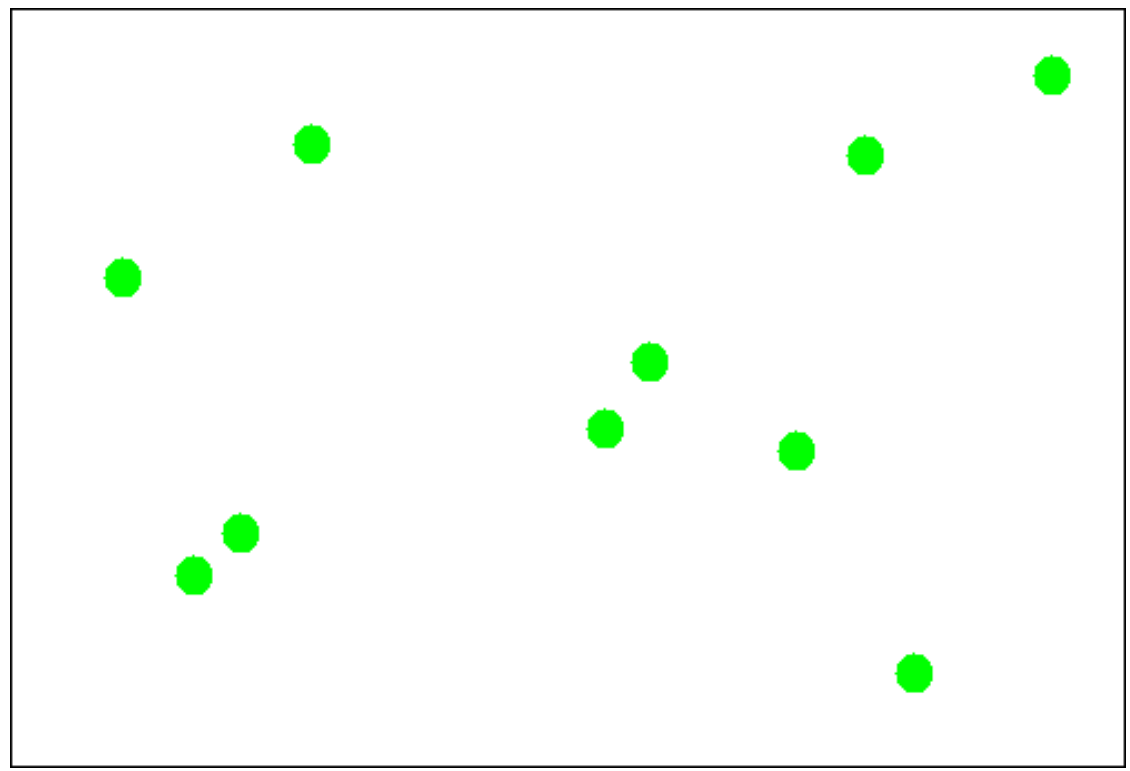
© Spinnaker Labs, Inc.



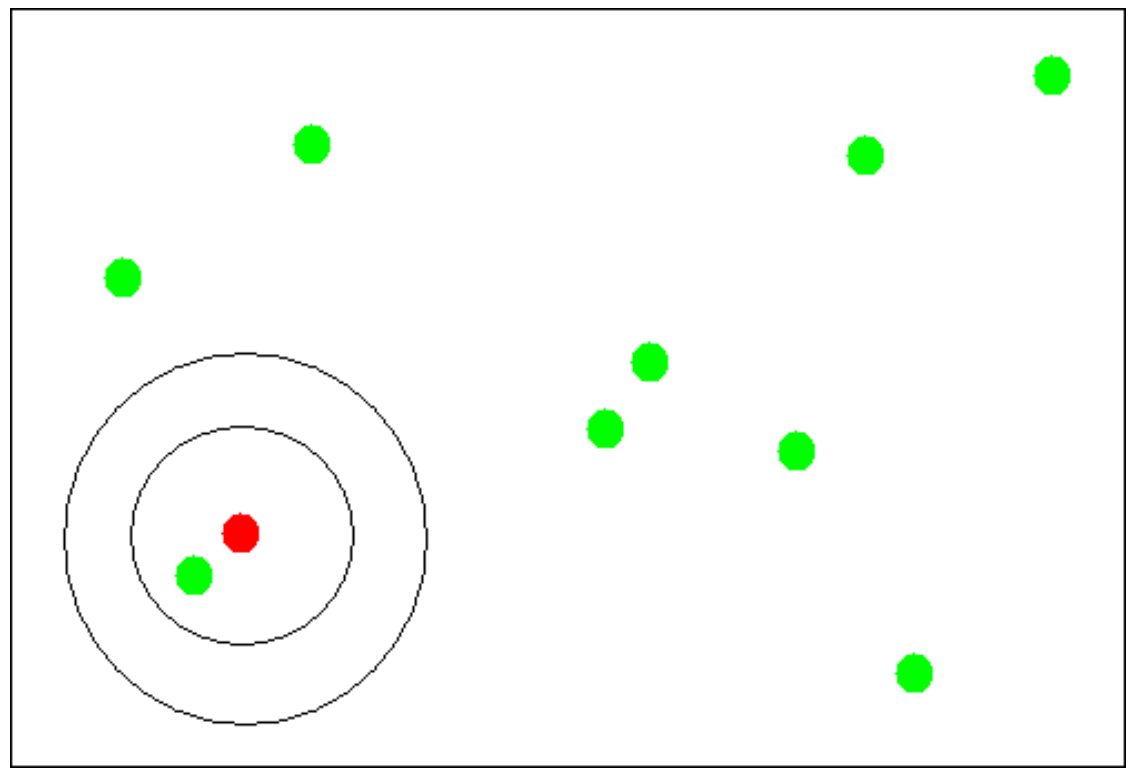
© Spinnaker Labs, Inc.



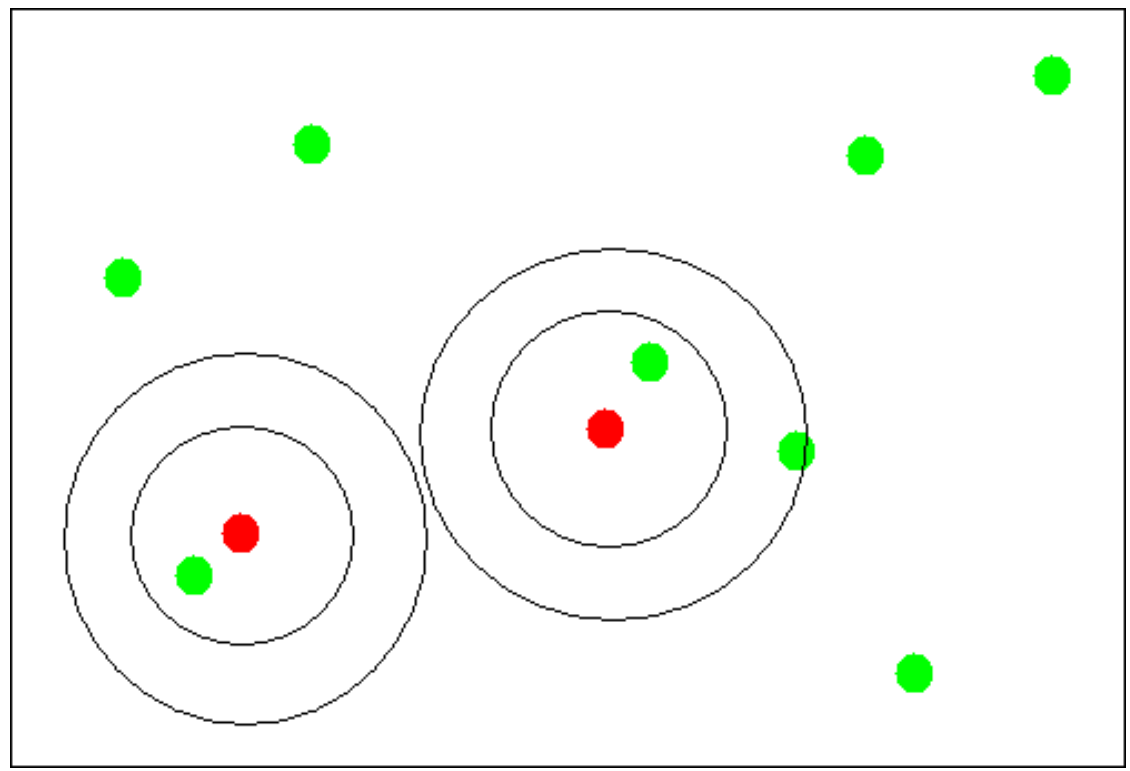
© Spinnaker Labs, Inc.



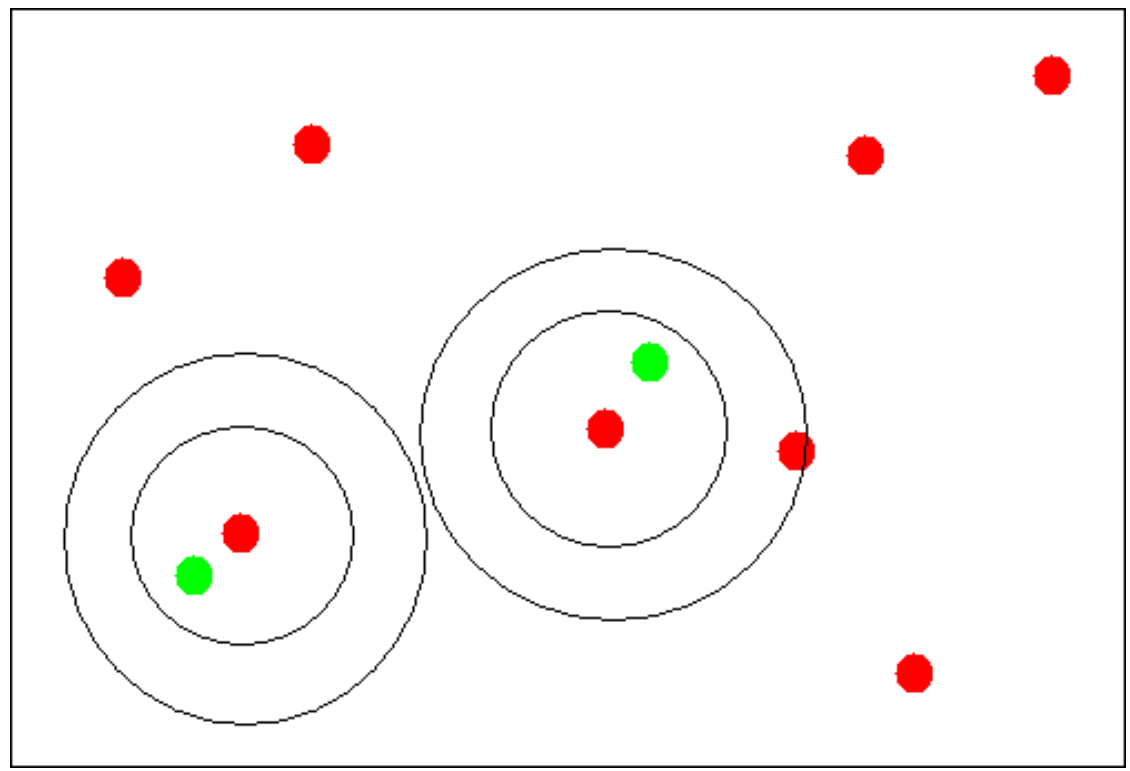
© Spinnaker Labs, Inc.



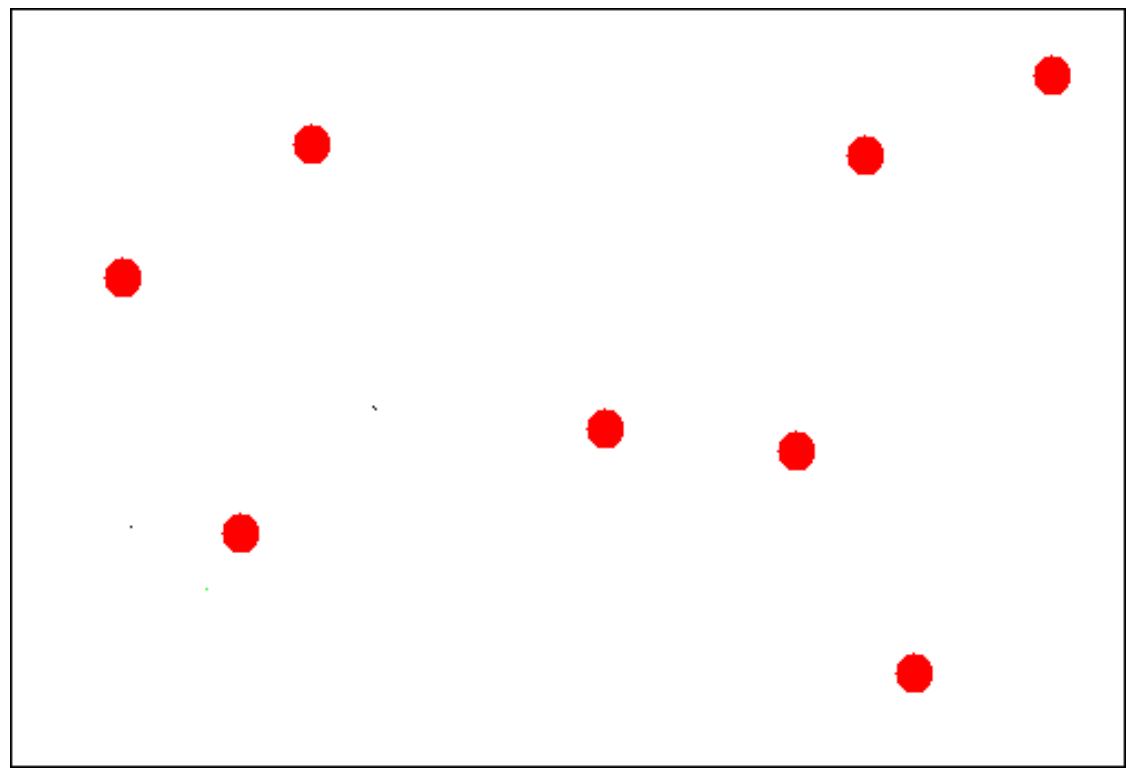
© Spinnaker Labs, Inc.



© Spinnaker Labs, Inc.



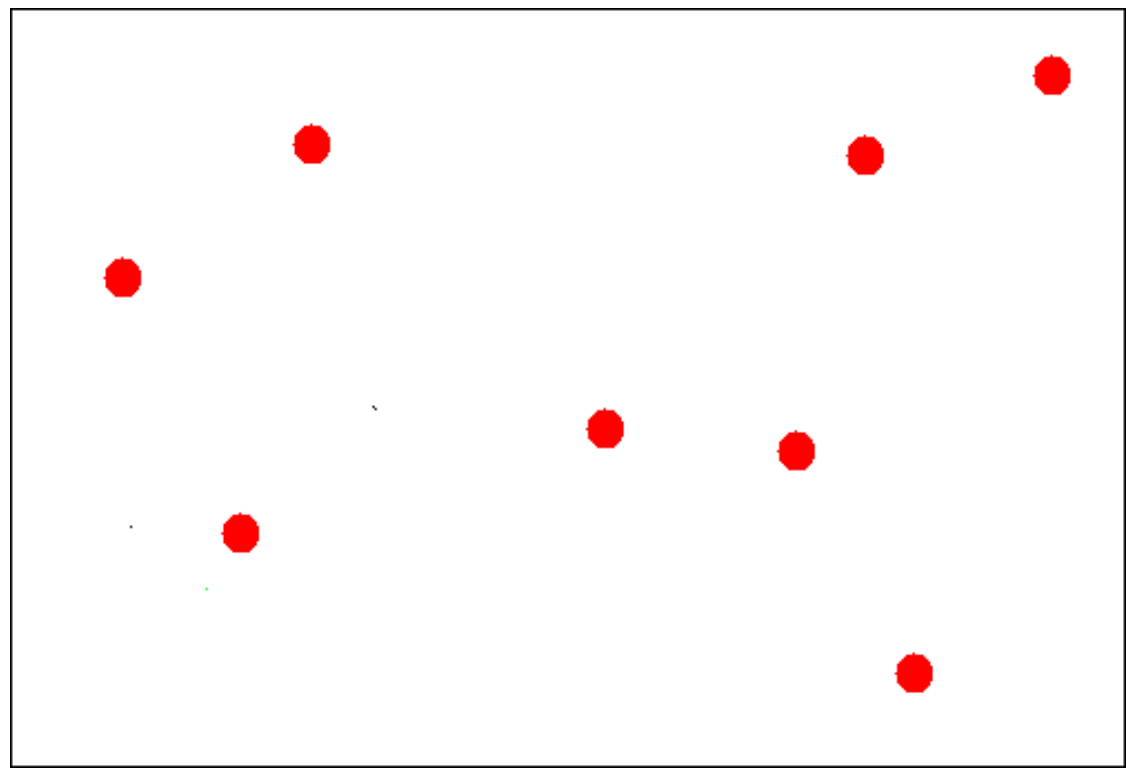
© Spinnaker Labs, Inc.



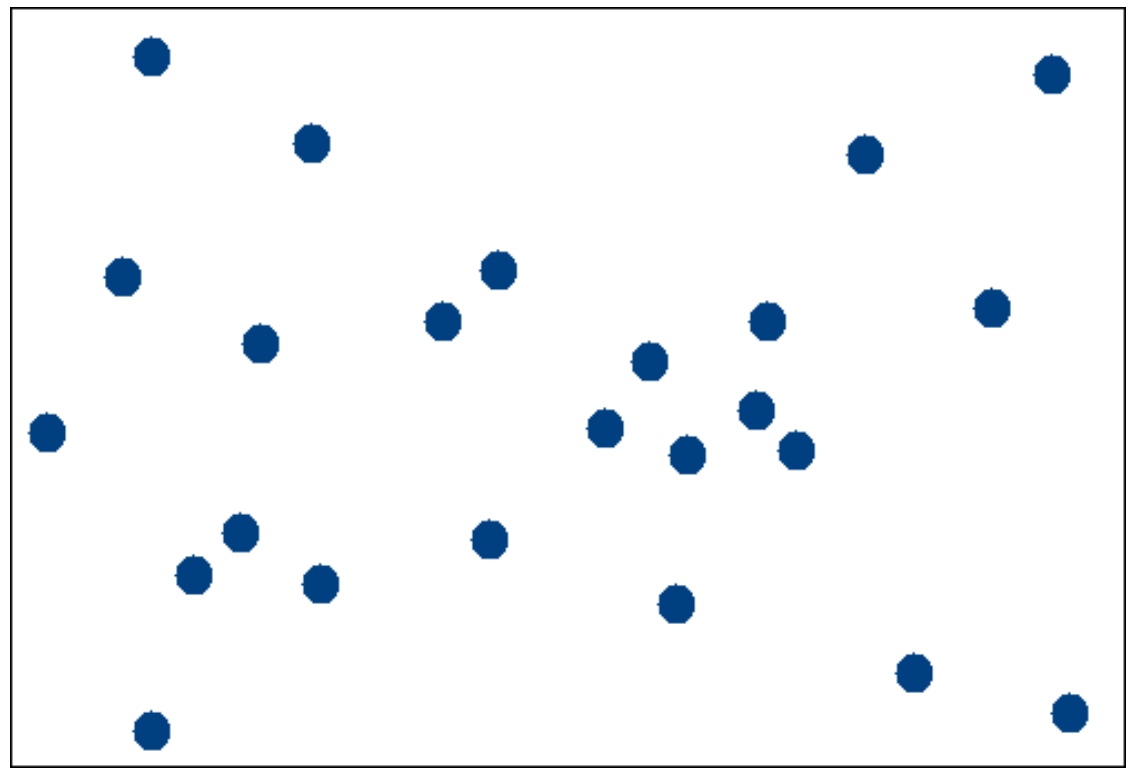
© Spinnaker Labs, Inc.

Assigning Points to Canopies

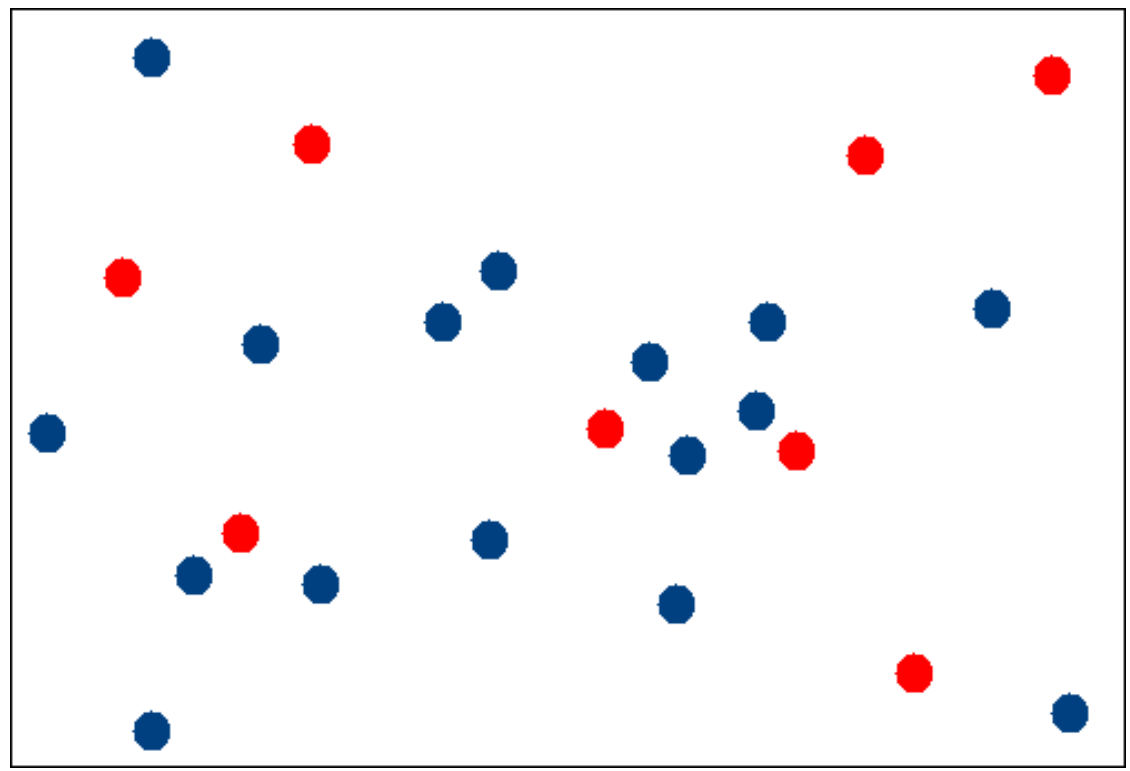
© Spinnaker Labs, Inc.



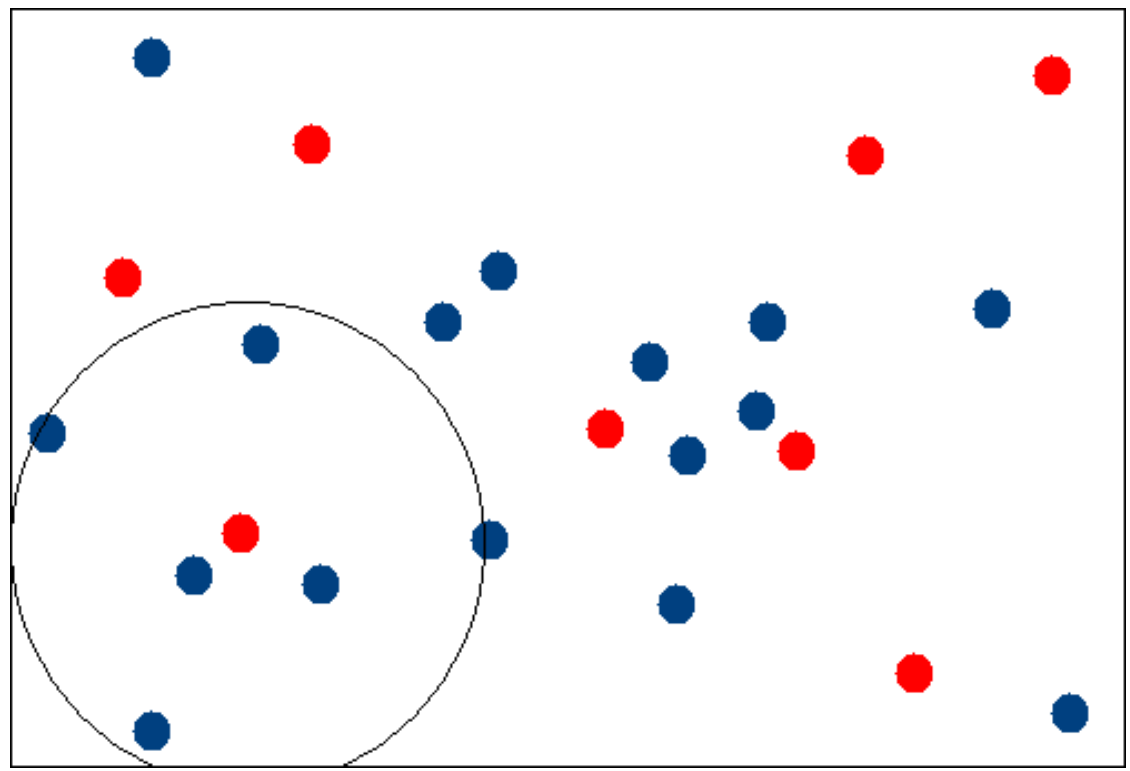
© Spinnaker Labs, Inc.



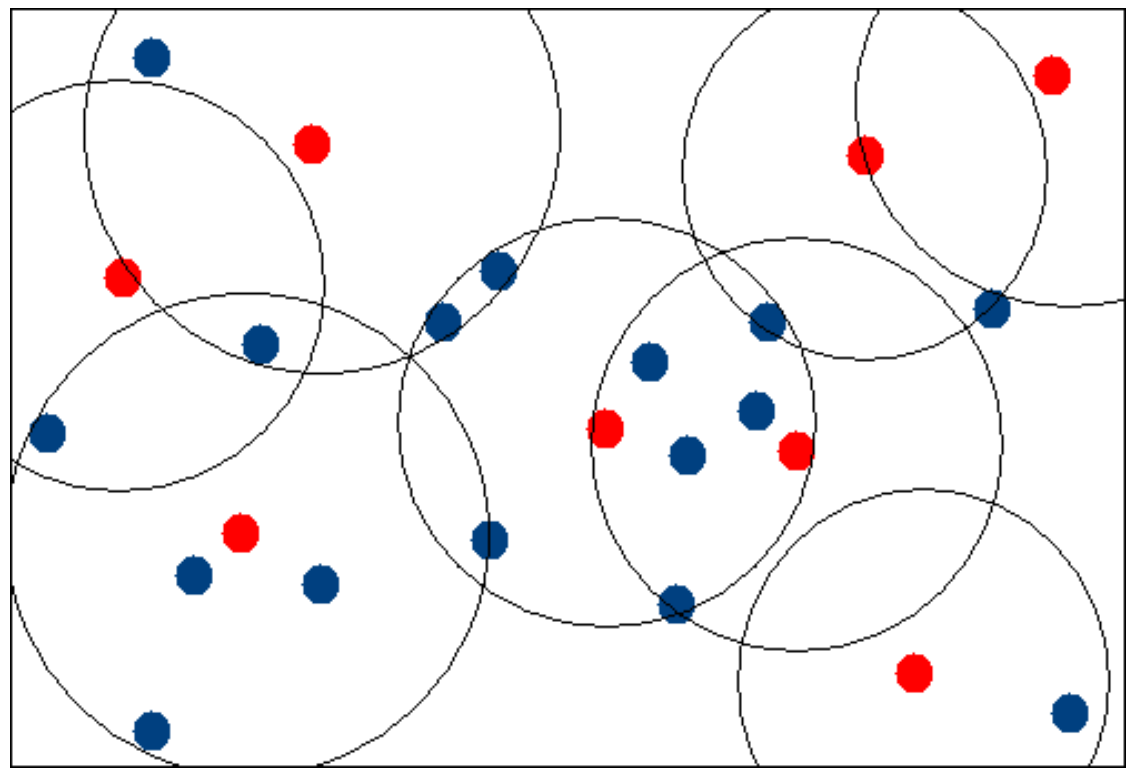
© Spinnaker Labs, Inc.



© Spinnaker Labs, Inc.



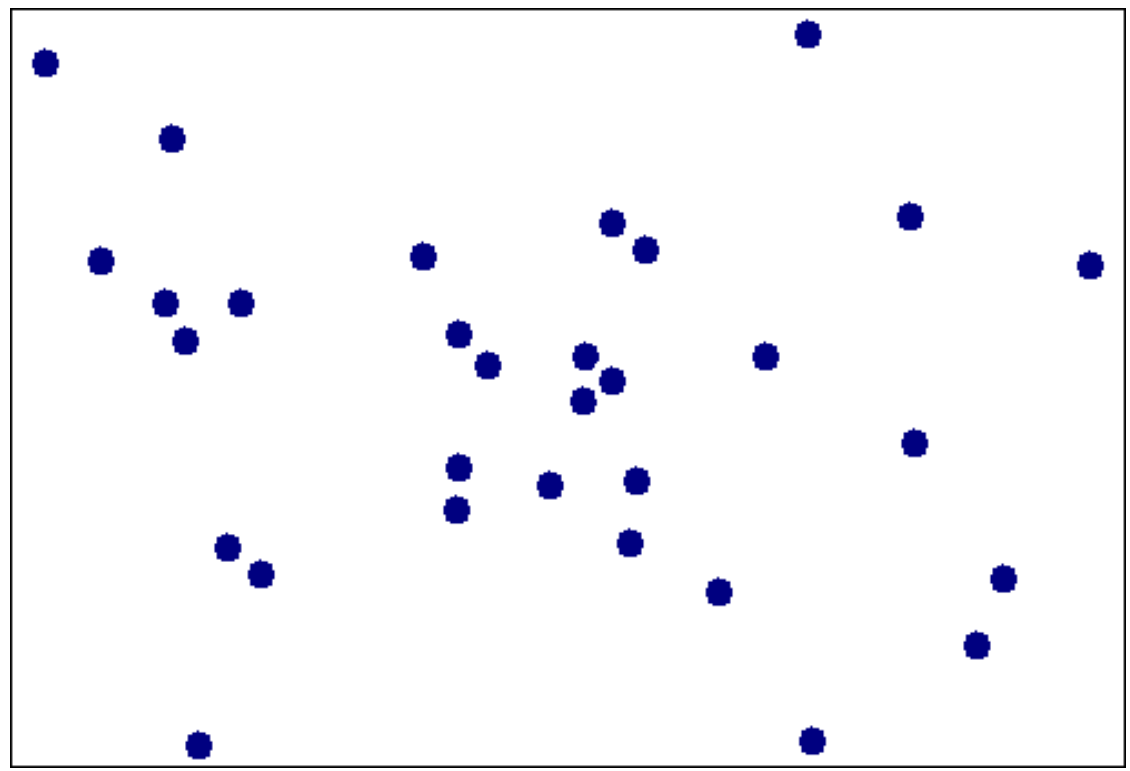
© Spinnaker Labs, Inc.



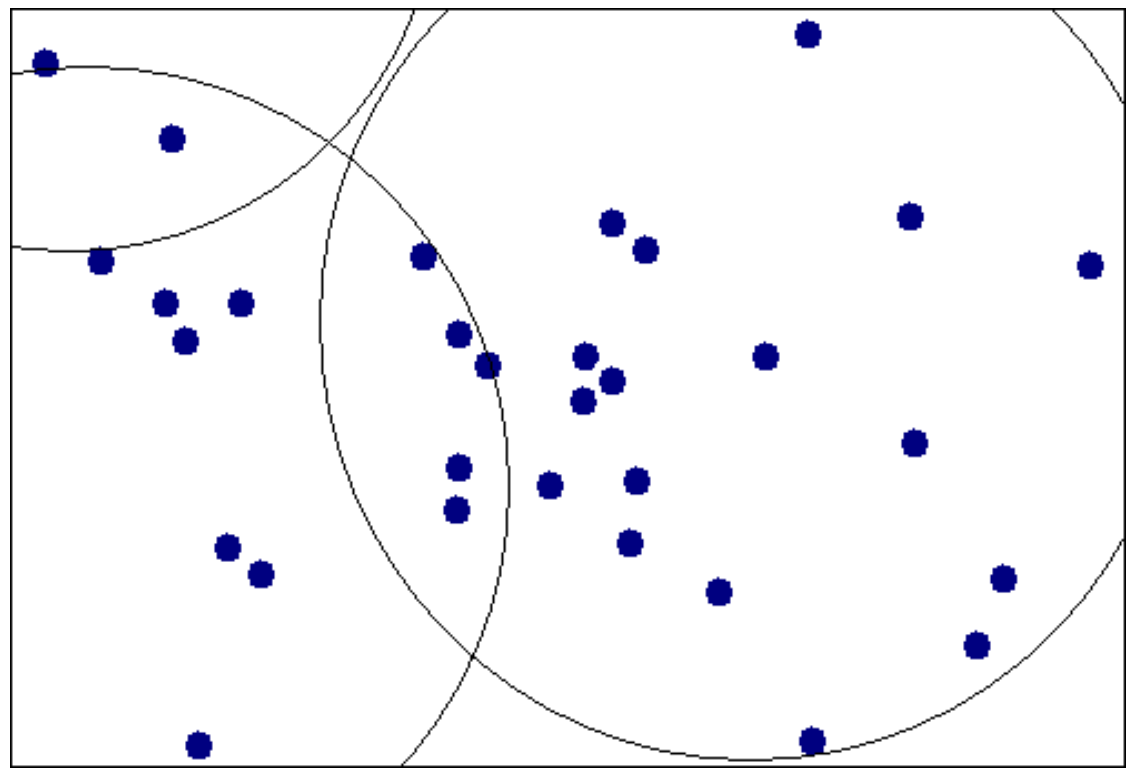
© Spinnaker Labs, Inc.

K-Means Map

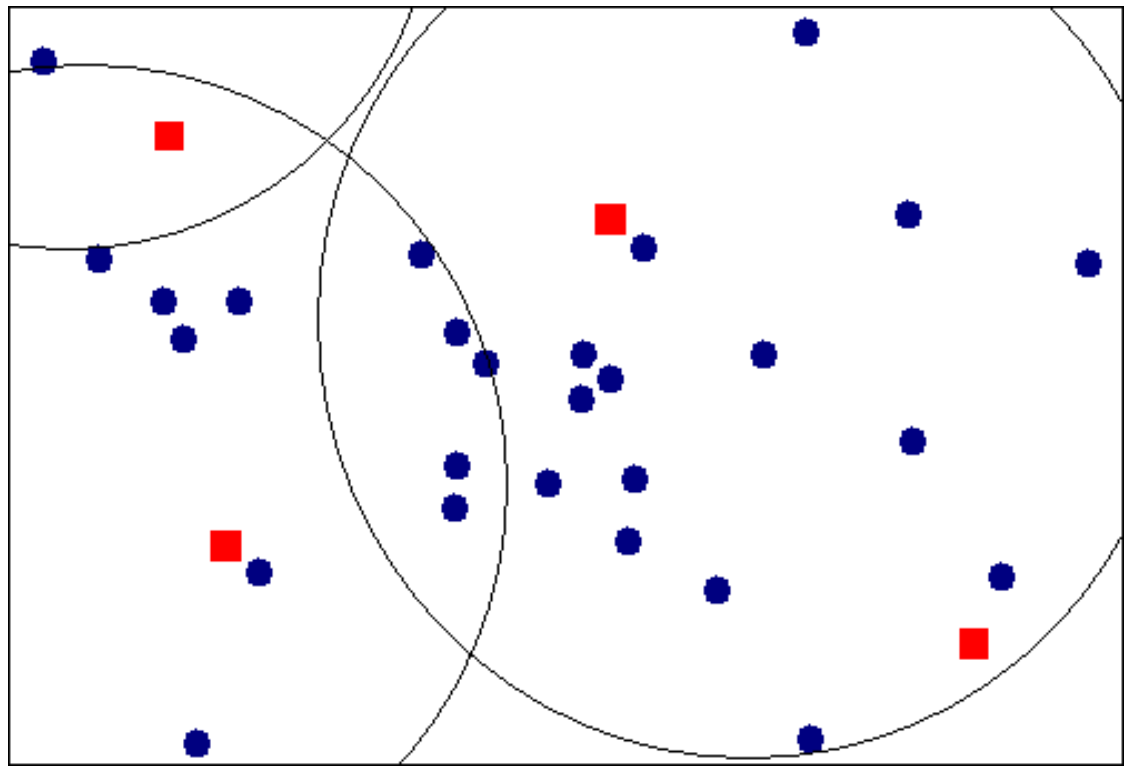
© Spinnaker Labs, Inc.



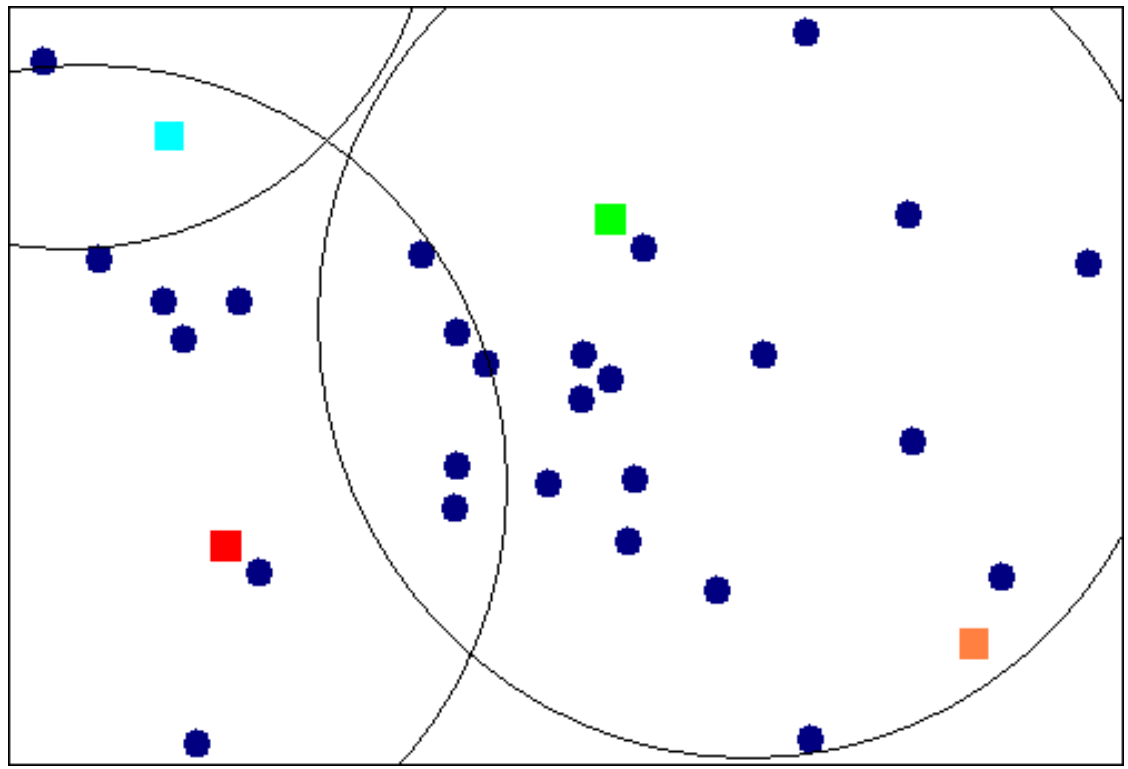
© Spinnaker Labs, Inc.



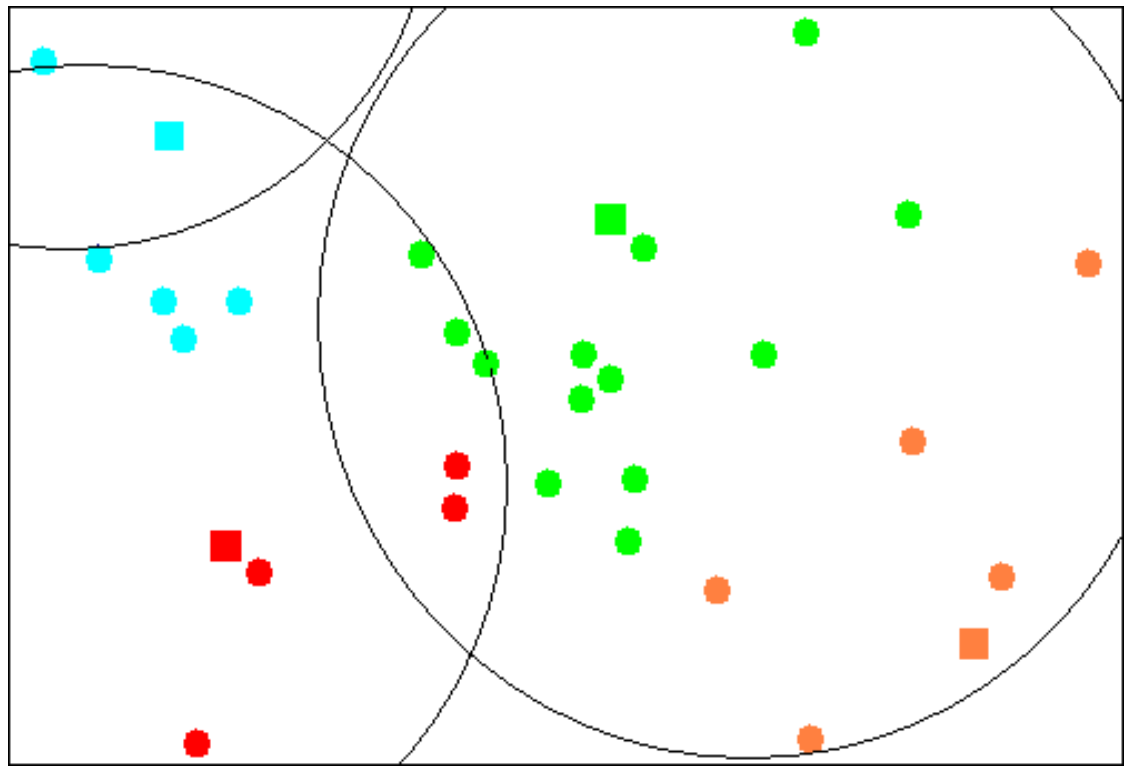
© Spinnaker Labs, Inc.



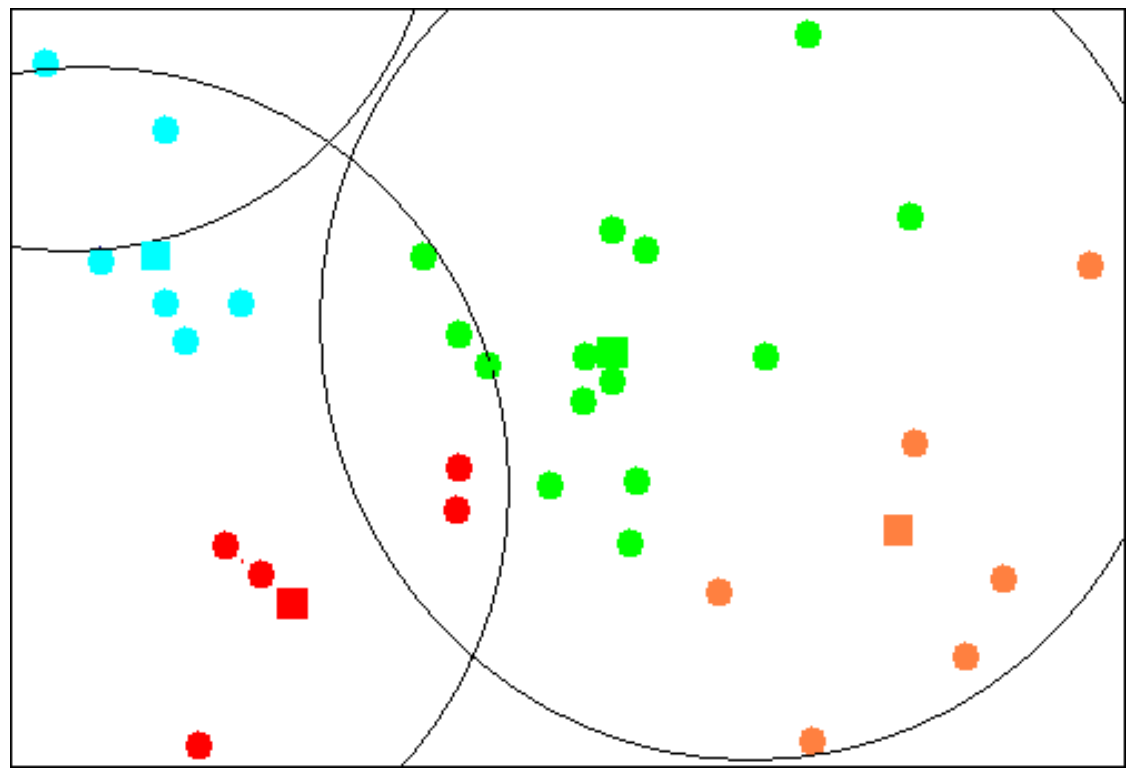
© Spinnaker Labs, Inc.



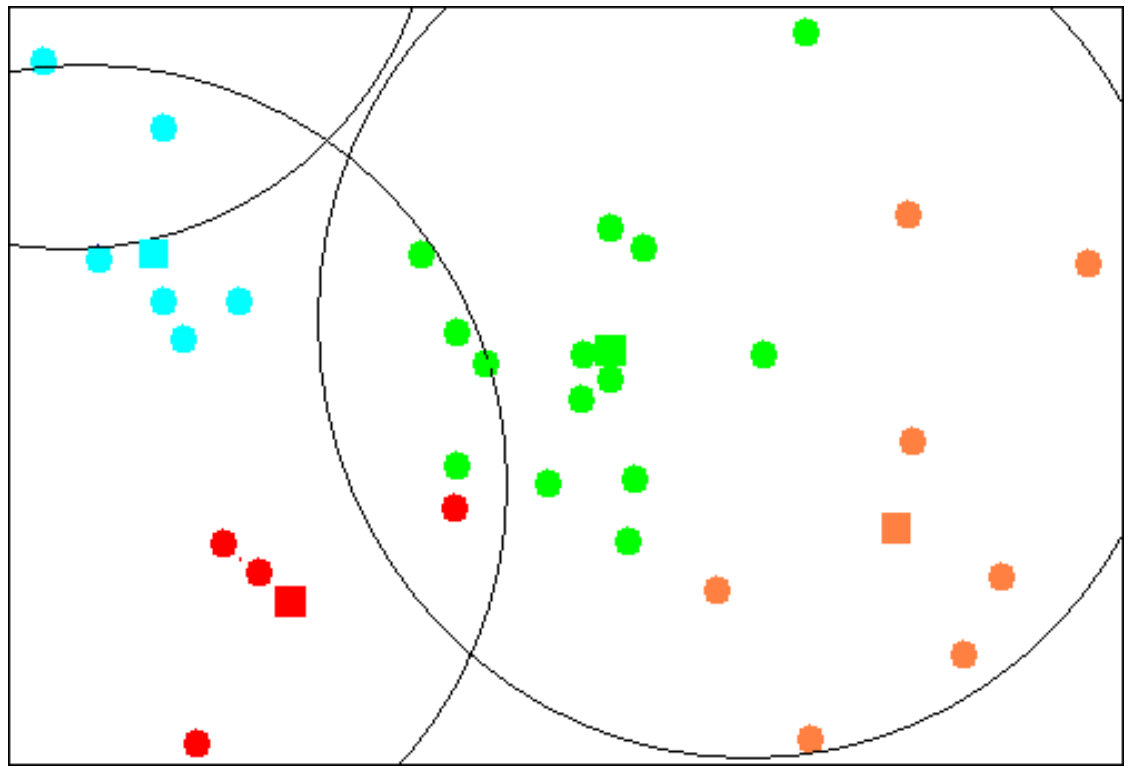
© Spinnaker Labs, Inc.



© Spinnaker Labs, Inc.



© Spinnaker Labs, Inc.



© Spinnaker Labs, Inc.

Elbow Criterion

- Choose a number of clusters s.t. adding a cluster doesn't add interesting information.

- Rule of thumb to determine what number of Clusters should be chosen.
- Initial assignment of cluster seeds has bearing on final model performance.
- Often required to run clustering several times to get maximal performance

© Spinnaker Labs, Inc.

Clustering Conclusions

- Clustering is slick
- And it can be done super efficiently
- And in lots of different ways

© Spinnaker Labs, Inc.

Overall Conclusions

- Lots of high level algorithms
- Lots of deep connections to low-level systems
- Clean abstraction layer for programmers between the two

© Spinnaker Labs, Inc.