Dynamic Programming, The Greedy Method

Applied Algorithms

Dynamic Programming

- Like the divide-and-conquer method, dynamic programming is used over specific computational problems.
- Dynamic programming can take problems that require exponential time and produces polynomial-time algorithms to solve them.
- Dynamic programming: use it when the subproblems overlap, e.g., Fibonacci numbers

Dynamic Programming

- Divide-and-conquer: use it when subproblems are disjoint.
    - partition the problem into disjoint subproblems
    - solve the subproblems recursively
    - combine their solutions to solve the original problem.
- Divide-and-conquer vs. Dynamic Programming ***when the subproblems overlap***:
    - A divide-and-conquer algorithm does more work than necessary—repeatedly solving the common subproblems.
    - A dynamic-programming algorithm solves each subproblem just once and then saves its answer in a table, so avoiding the work of recomputing the answer every time it solves each subproblem.

Dynamic Programming

- When developing a dynamic-programming algorithm, we follow a sequence of four steps:

    1. Characterize the structure of an optimal solution.

    2. Recursively define the value of an optimal solution.

    3. Compute the value of an optimal solution, typically in a bottom-up fashion.

    4. Construct an optimal solution from computed information.

- When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

- Example: https://www.youtube.com/watch?v=e0CAbRVYAWg

Dynamic Programming : Matrix Chain-Product

- Suppose we are given a collection of n two-dimensional matrices for which we wish to compute the mathematical product $A = A_0 \cdot A_1 \cdot A_2 \ \ldots \ A_{n-1}$ where A_i is a $d_i \times d_{i+1}$matrix, for $i = 0,1,2,\ldots,n-1..$ In the standard matrix multiplication algorithm to multiply a $d \times e$-matrix B times an $e \times f$-matrix C, we compute the product, A, as.

$$A[i][j] \sum_{k=0}^{e-1} B[i][k] \cdot C[k][j]$$

Dynamic Programming : Matrix Chain-Product

- We can multiply two matrices A and B only if they are compatible:

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2       error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5            for j = 1 to B.columns
6                 c_ij = 0
7                 for k = 1 to A.columns
8                      c_ij = c_ij + a_ik · b_kj
9       return C
```

Dynamic Programming : Matrix Chain-Product

- $B \cdot (C \cdot D) = (B \cdot C) \cdot D$—associative
- The matrix chain-product problem is to determine the parenthesization of the expression defining the product A that minimizes the total number of scalar multiplications performed.
- Example: Let B, C, and D be a 3×30, 30×80, 80×30 -matrices. How many multiplication required to calculate $B \cdot (C \cdot D)$ and $(B \cdot C) \cdot D$?
- Example:

$$
\begin{aligned}
&(A_1(A_2(A_3A_4))) \ , \\
&(A_1((A_2A_3)A_4)) \ , \\
&((A_1A_2)(A_3A_4)) \ , \\
&((A_1(A_2A_3))A_4) \ , \\
&(((A_1A_2)A_3)A_4) \ .
\end{aligned}
$$

Brute-Force

- Matrix-chain multiplication problem:

  Given a chain $< A_1, A_2 \ldots, A_n >$ of n matrices, where for $i = 1,2,\ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$ fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

- Brute-force: enumerate all the possible ways of parenthesizing the expression and determine the number of multiplications performed by each one.

- What is wrong with this?

  The number of solutions is exponential in n.

Applying dynamic programming

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from computed information.

Applying dynamic programming: Step 1 - The structure of an optimal parenthesization

- Find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems
- $A_i \cdot A_{i+1} \cdot A_j, i \leq j$
- $A_{i \ldots k}, A_{k+1 \ldots j}$
- The optimal substructure of this problem is as follows:

Applying dynamic programming: Step 1 - The structure of an optimal parenthesization

Suppose that to optimally parenthesize $A_i A_{i+1} \ldots A_j$. We split the product between $A_k$and A_{k+1}. Then the way we parenthesize the "prefix" subchain $A_i A_{i+1} \ldots A_k$ within this optimal parenthesization of $A_i A_{i+1} \ldots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \ldots A_k$.

Applying dynamic programming: Step 1 - The structure of an optimal parenthesization

- Any solution to a nontrivial instance of the matrix-chain multiplication problem requires us to split the product, and that any optimal solution contains within it optimal solutions to subproblem instances.
- Build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions.
- When searching for the correct place to split the product, we should consider all possible places and return the optimal one.

Applying dynamic programming
Step 2: A recursive solution

- $A_i A_{i+1} \ldots A_j$, and $1 \leq i \leq j \leq n$.
- Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i \ldots j}$
- The lowest cost way to compute $A_{1 \ldots n}$ for the full problem: is $m[1,n]$.

Applying dynamic programming
Step 2: A recursive solution

1. Define $m[i, j]$ recursively as follows:

   Each matrix A_i is $p_{i-1} \times p_i$, computing the matrix product $A_{i \ldots k} A_{k+1 \ldots j}$ takes $p_{i-1} p_k p_j$ scaler multiplications.

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

2. There are only j-i possible values for k. Since the optimal parenthesization must use one of these values for k, we need only check them all to find the best.

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

Applying dynamic programming
Step 2: A recursive solution

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \ldots A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

Applying dynamic programming: Step 3: Computing the optimal costs

There have relatively few distinct subproblems: one subproblem for each choice of i and j satisfying

$$1 \le i \le j \le n, \text{ or } \binom{n}{2} + n = \theta(n^2) \text{ in all.}$$

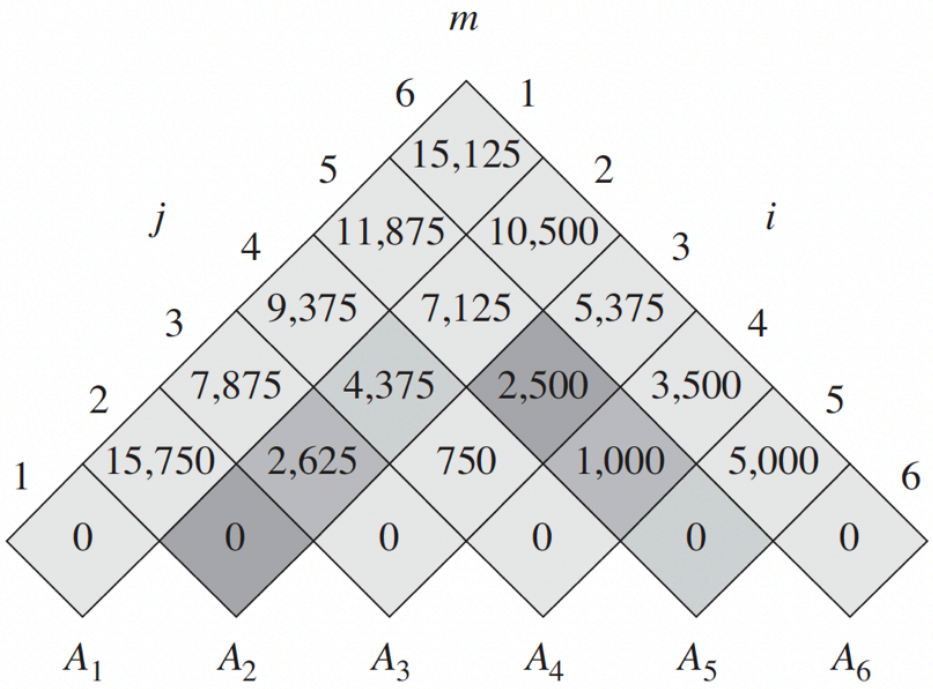
Applying dynamic programming: Step 3: Computing the optimal costs

- Instead of computing the solution to recurrence recursively, compute the optimal cost by using a tabular, bottom-up approach
- $A_i, p_{i-1} \times p_i, i = 1,\ldots,n$ matrix, its input is a sequence $p = < p_0, p_1, \ldots, p_n > , p\,.\,length = n + 1$
- Auxiliary tables:
  - $m[1..n,1..n]$ — storing $m[i, j]$ costs
  - $s[1..n-1,2..n]$ — records which index of k achieved the optimal cost in computing $m[i, j]$

Applying dynamic programming: Step 3: Computing the optimal costs

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

Applying dynamic programming: Step 3: Computing the optimal costs

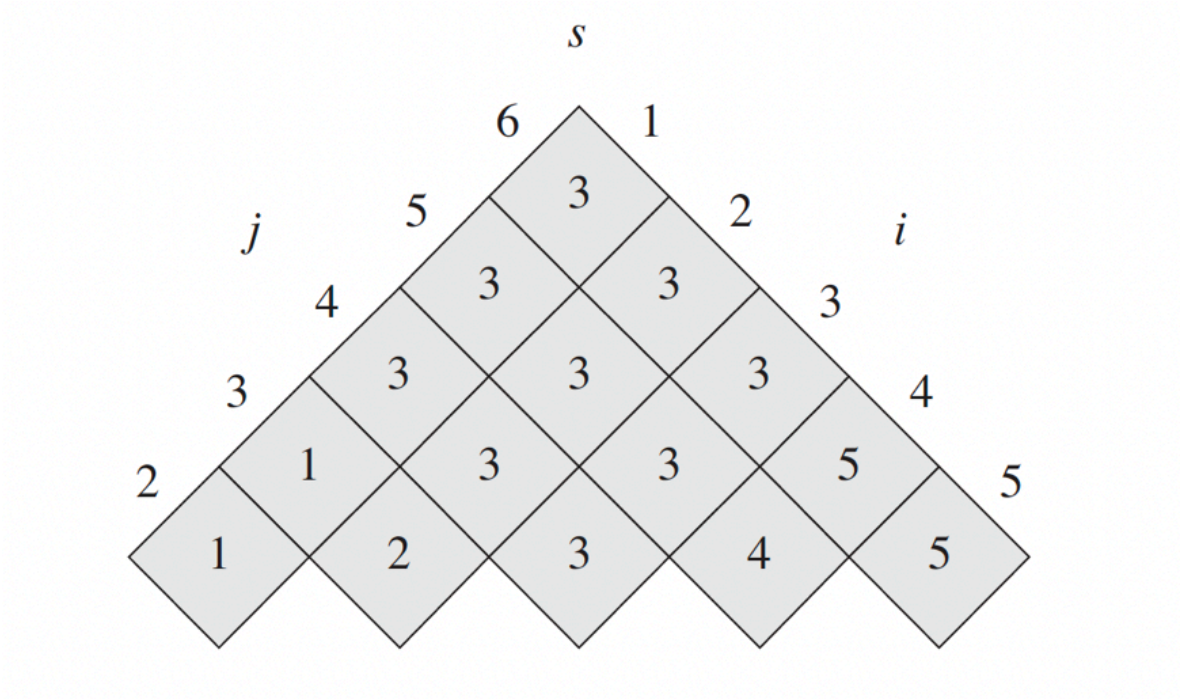


matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125\,.$$

The minimum number of scalar multiplications to multiply the 6 matrices is m[1,6] = 15,125

Applying dynamic programming: Step 4: Constructing an optimal solution



$$((A_1(A_2A_3))((A_4A_5)A_6))$$

Applying dynamic programming:
Step 4: Constructing an optimal solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

Applying dynamic programming:

1. https://www.youtube.com/watch?v=P8Xa2BitN3I

2.https://www.youtube.com/watch?v=_zE5z-KZGRw

The Greedy Method

- Construct some structure while minimizing or maximizing some property of that structure.
- Proceed to the solution by a sequence of choices.
- Start from some well-understood starting condition, and computes the cost for that initial condition.
- Iteratively make additional choices by identifying the decision that achieves the best cost improvement from all of the choices that are currently possible.
- This approach does not always lead to an optimal solution.

The Greedy Method

- Optimal solution and greedy-choice property: A global optimal condition can be reached by a series of locally optimal choices starting from a well-defined starting condition.

- Example: The problem of computing an optimal variable-length prefix code.

The Greedy Method: Text Compression

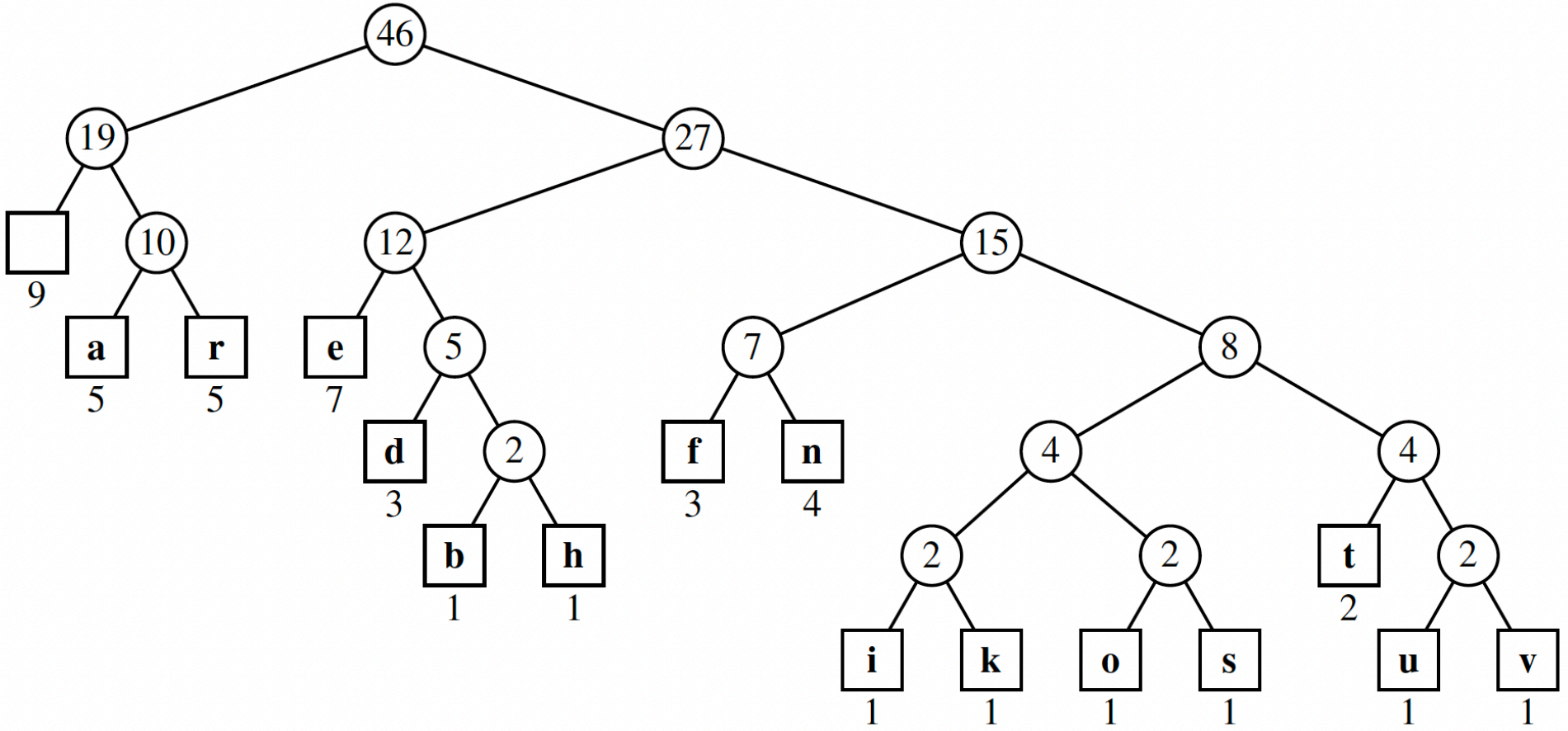
- Th Huffman's algorithm is an example application of an algorithmic design pattern called the greedy method.
- Use short code-word strings to encode high-frequency characters and long code-word strings to encode low-frequency characters.
- Huffman's algorithm constructs an optimal prefix code for a string of length n with d distinct characters in O(n+d logd) time.
- Video: https://www.youtube.com/watch?v=dM6us854Jk0
- Video: https://www.youtube.com/watch?v=0B_Ocir2maU

Huffman code: Class Exercise

X = "a fast runner need never be afraid of the dark”

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

Huffman code: Class Exercise



The Huffman Coding Algorithm

Algorithm Huffman(X):

Input: String X of length n with d distinct characters

Output: Coding tree for X

Compute the frequency $f(c)$ of each character c of X.

Initialize a priority queue Q.

for each character c in X **do**

Create a single-node binary tree T storing c.

Insert T into Q with key $f(c)$.

while len(Q) > 1 **do**

$(f_1, T_1) = Q$.remove_min()

$(f_2, T_2) = Q$.remove_min()

Create a new binary tree T with left subtree T_1 and right subtree T_2.

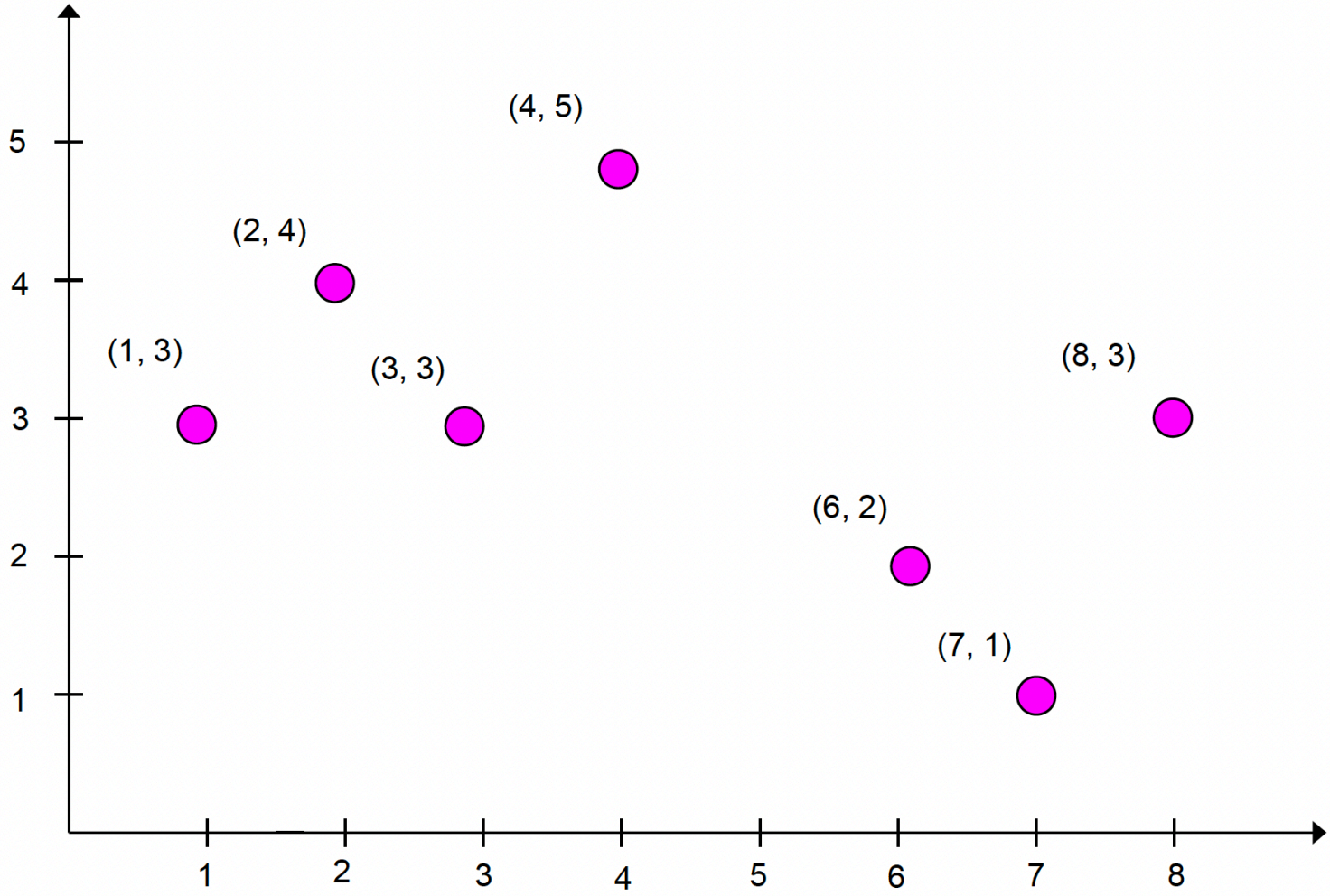
Insert T into Q with key $f_1 + f_2$.

$(f, T) = Q$.remove_min()

return tree T

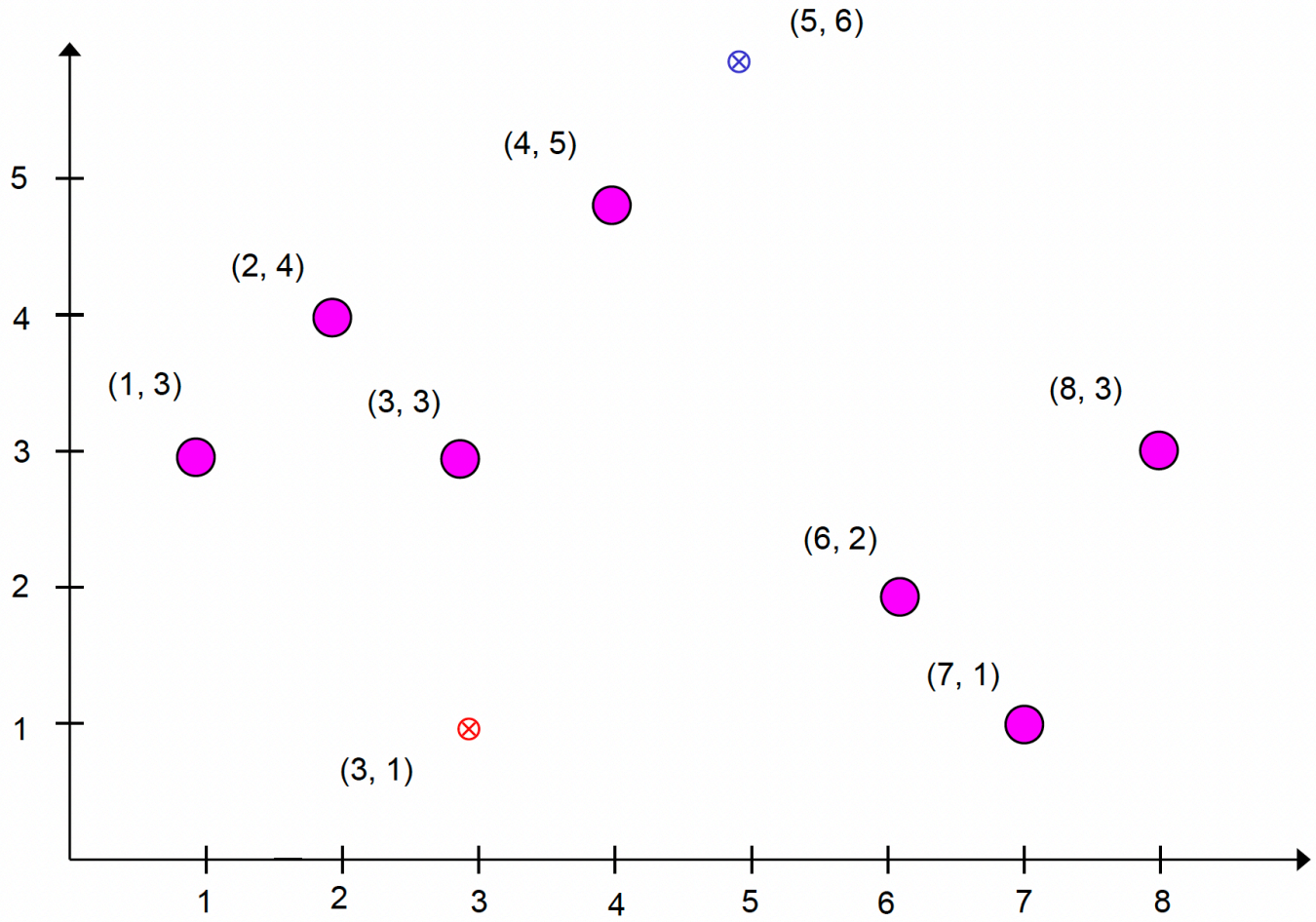
The Greedy Method: Clustering

K-means: Example



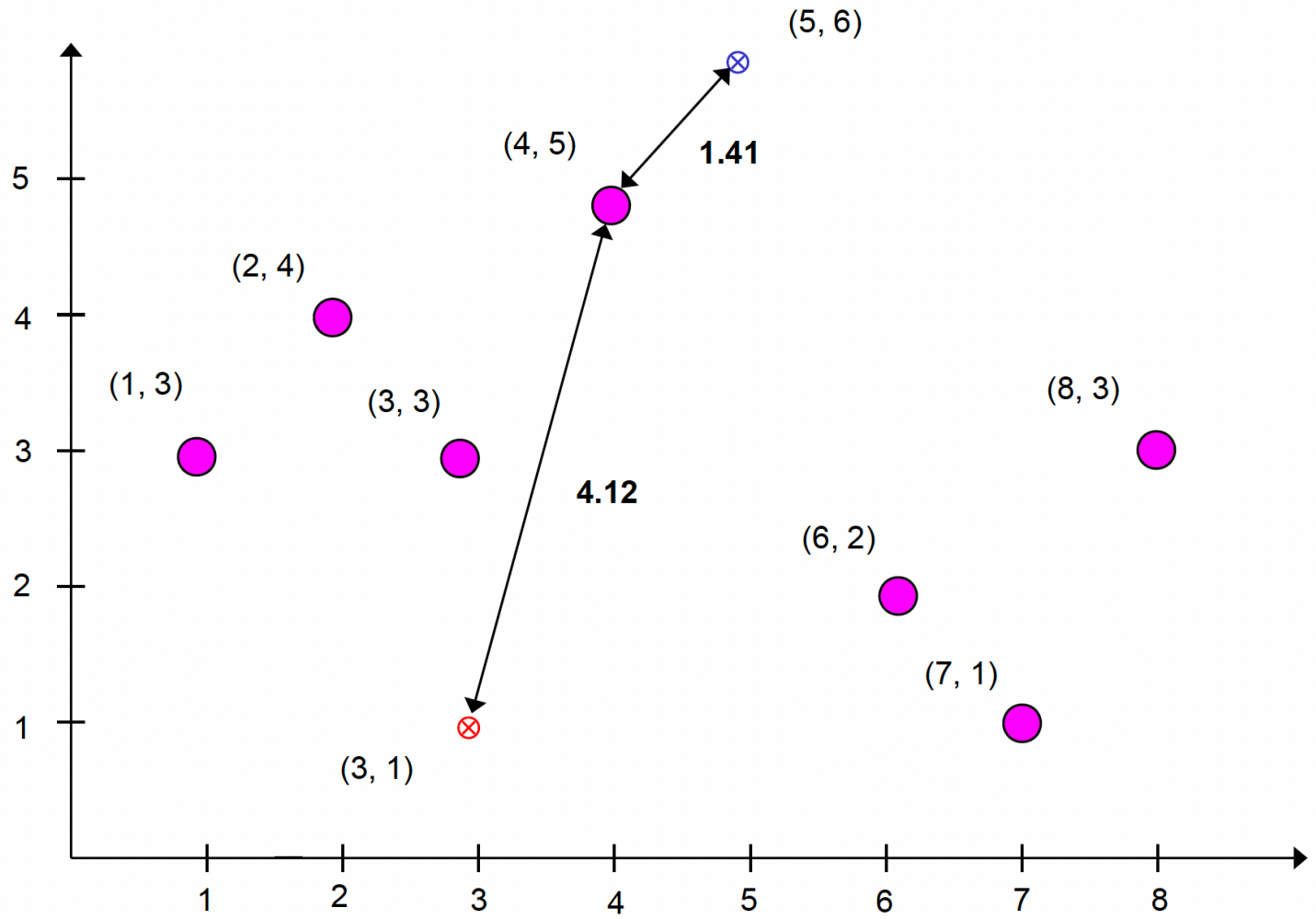
The Greedy Method: Clustering

K-means: Example



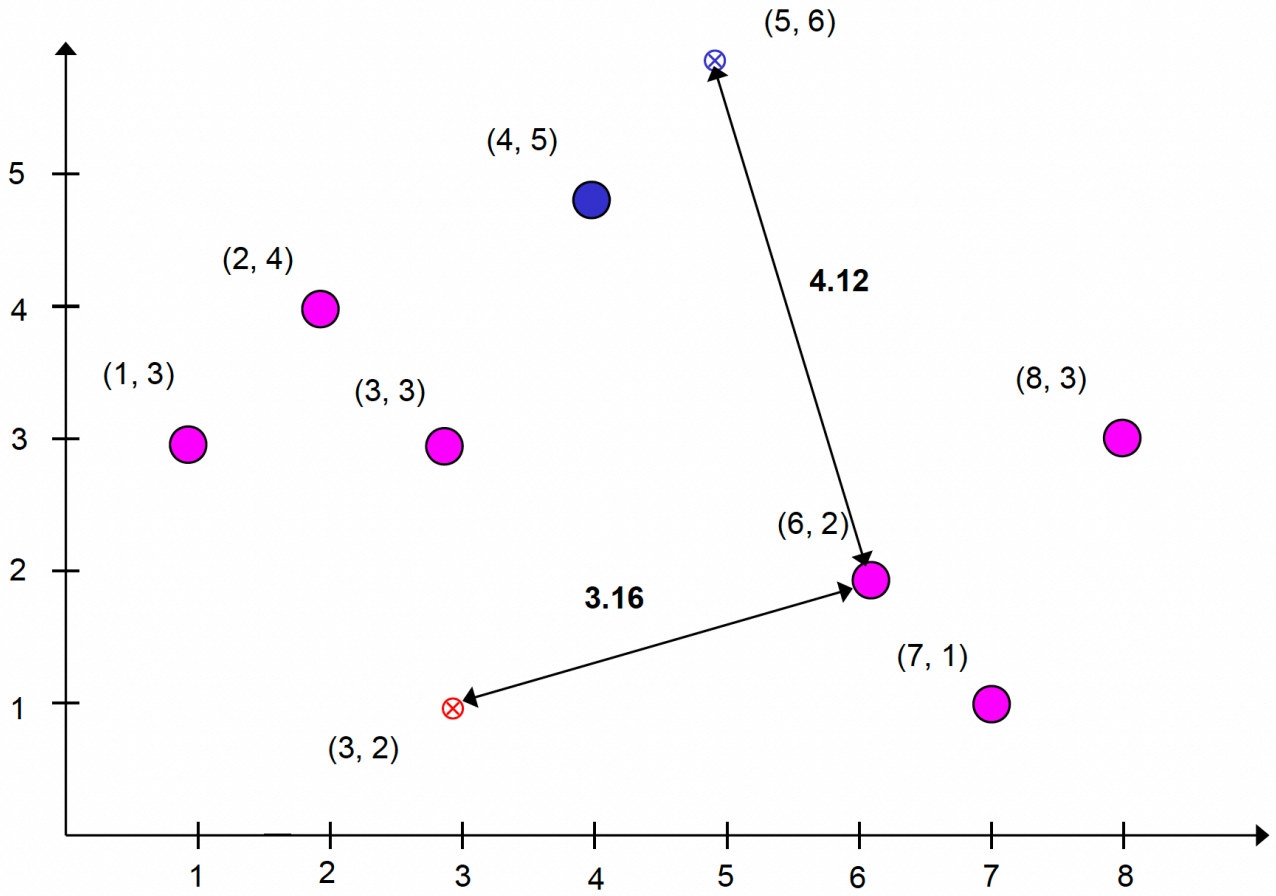
The Greedy Method: Clustering

K-means: Example



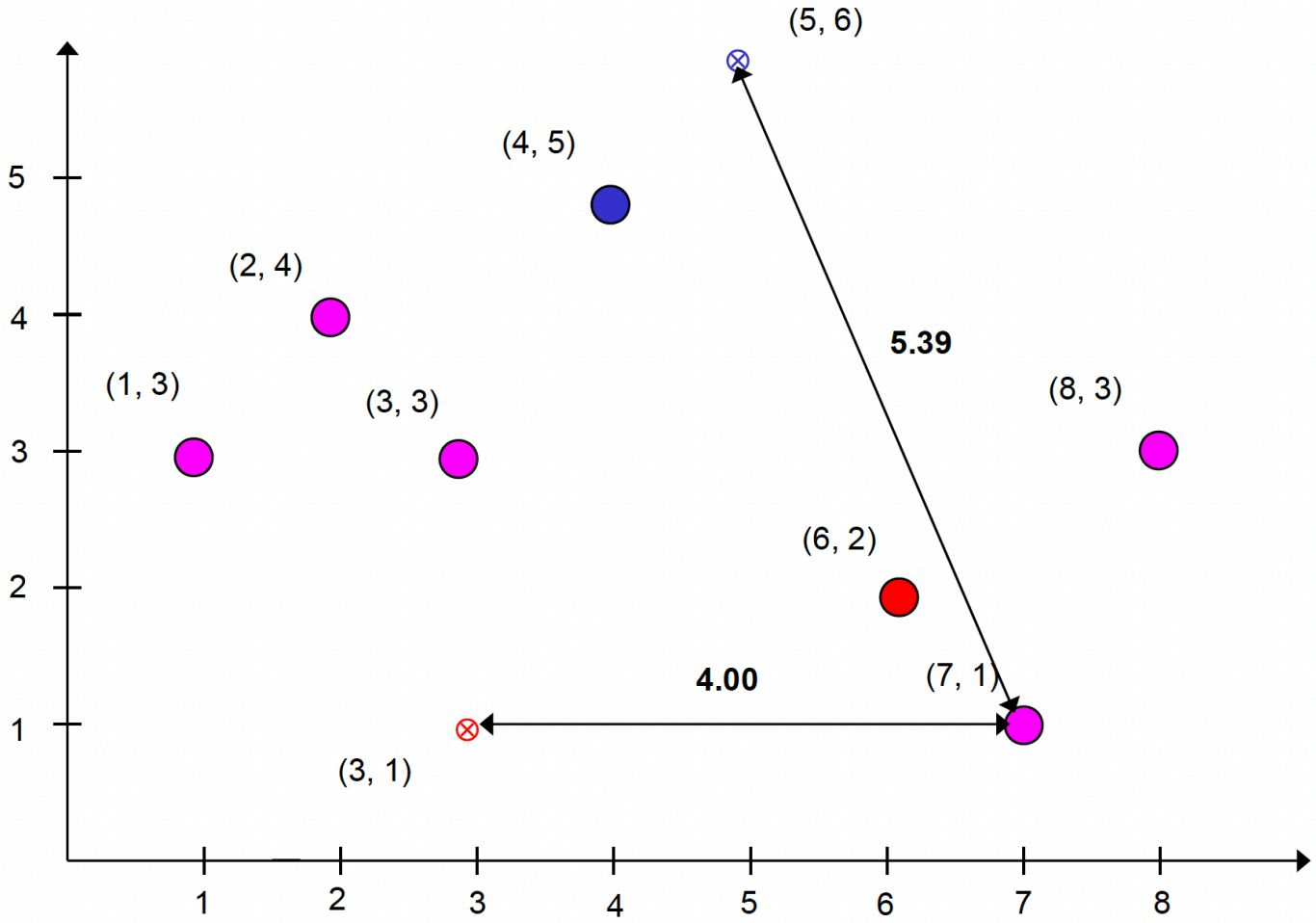
The Greedy Method: Clustering

K-means: Example



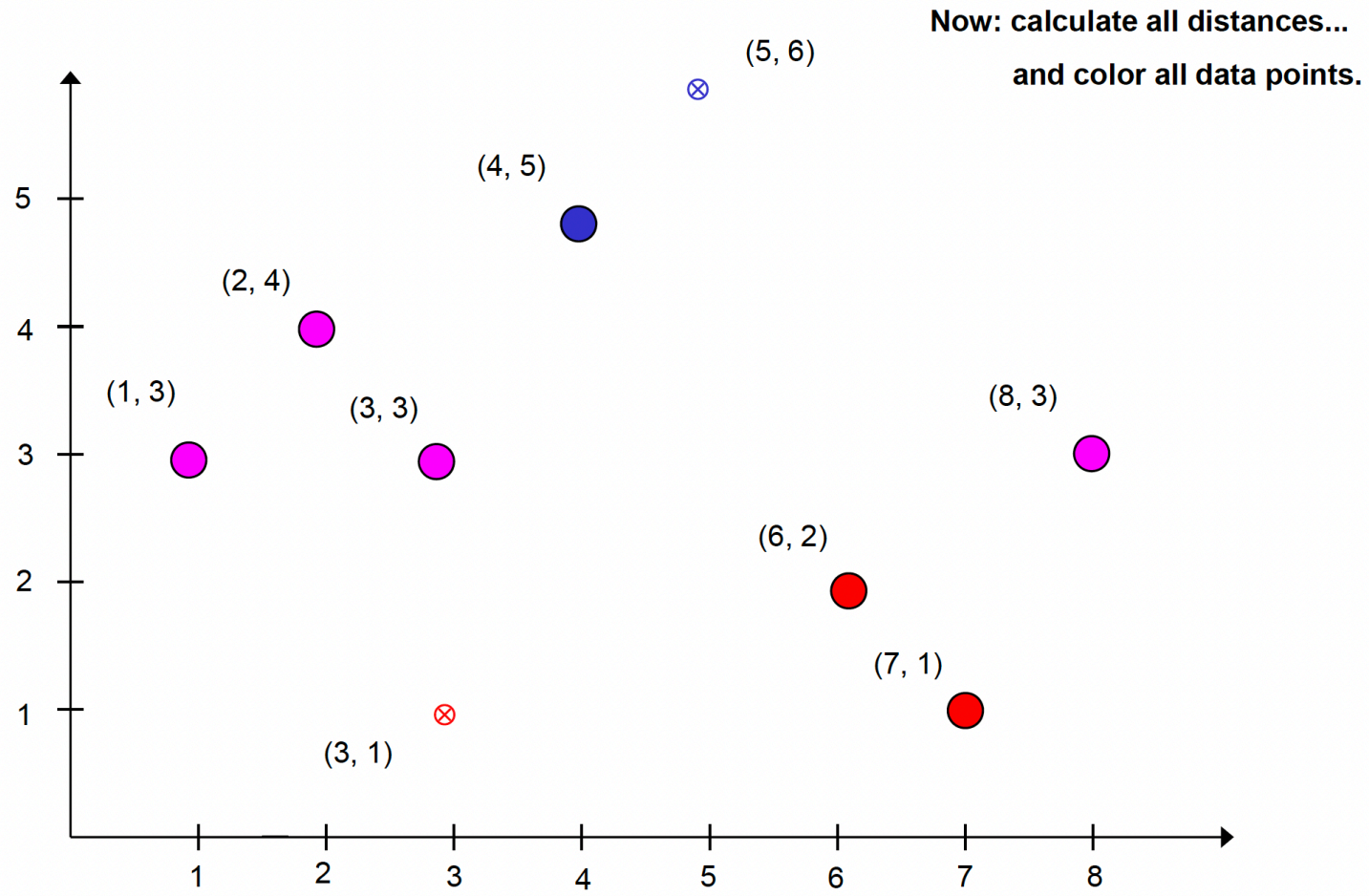
The Greedy Method: Clustering

K-means: Example



The Greedy Method: Clustering

K-means: Example



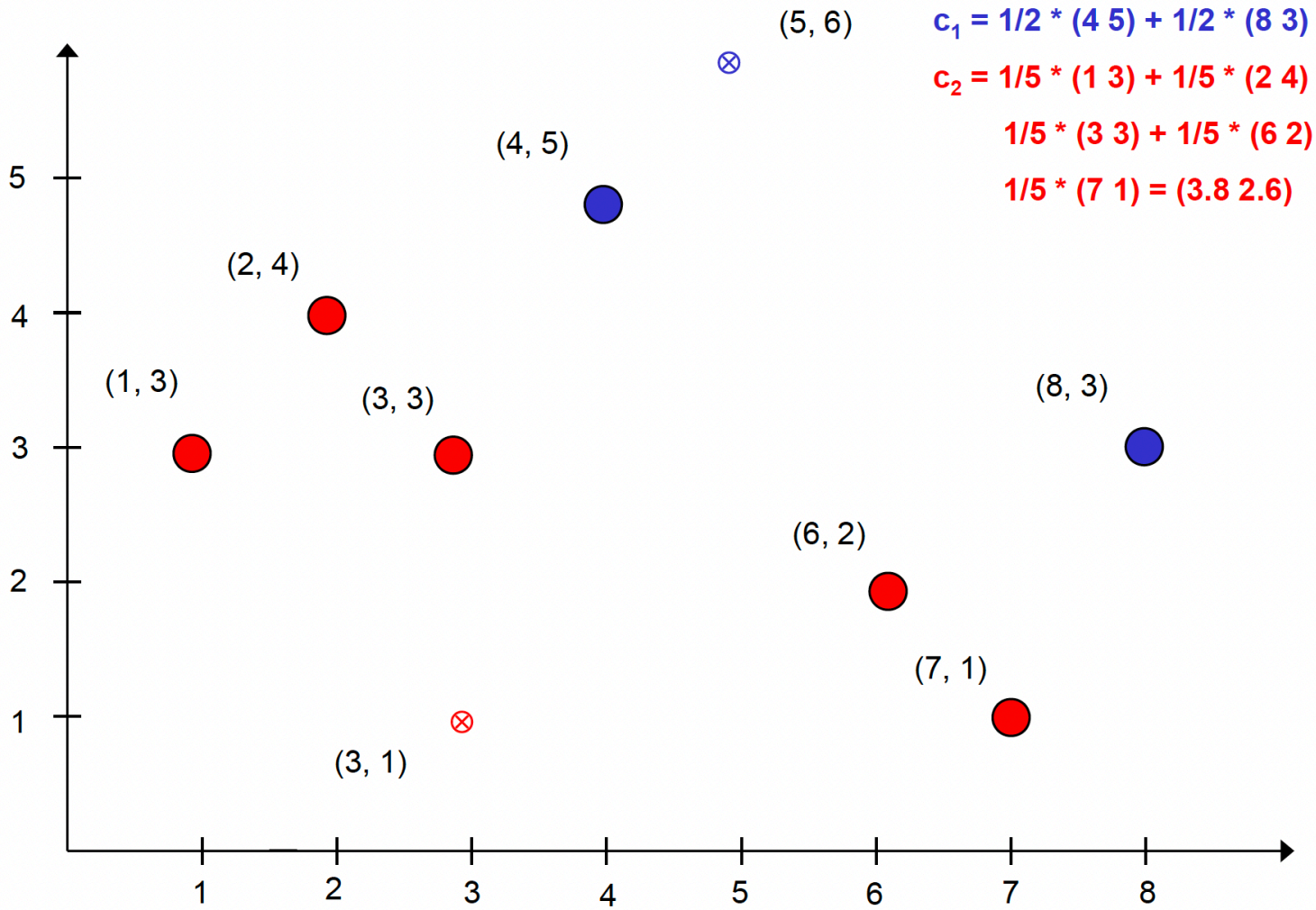
The Greedy Method: Clustering

K-means: Example

Now: move cluster centers to be the average of data points.

c_1 = 1/2 * (4 5) + 1/2 * (8 3) = (6 4)

c_2 = 1/5 * (1 3) + 1/5 * (2 4) +
1/5 * (3 3) + 1/5 * (6 2) +
1/5 * (7 1) = (3.8 2.6)

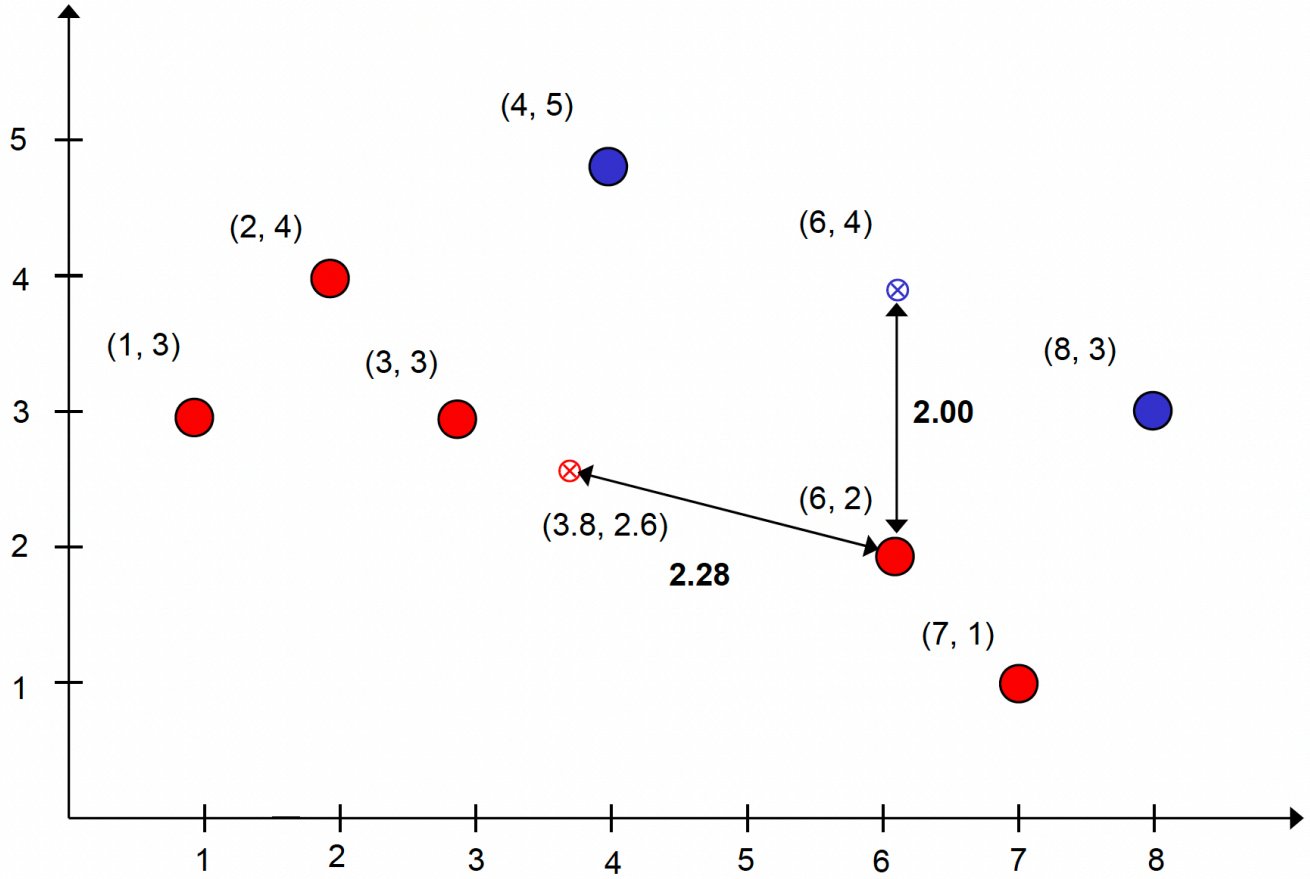


The Greedy Method: Clustering

K-means: Example

Now: start next iteration

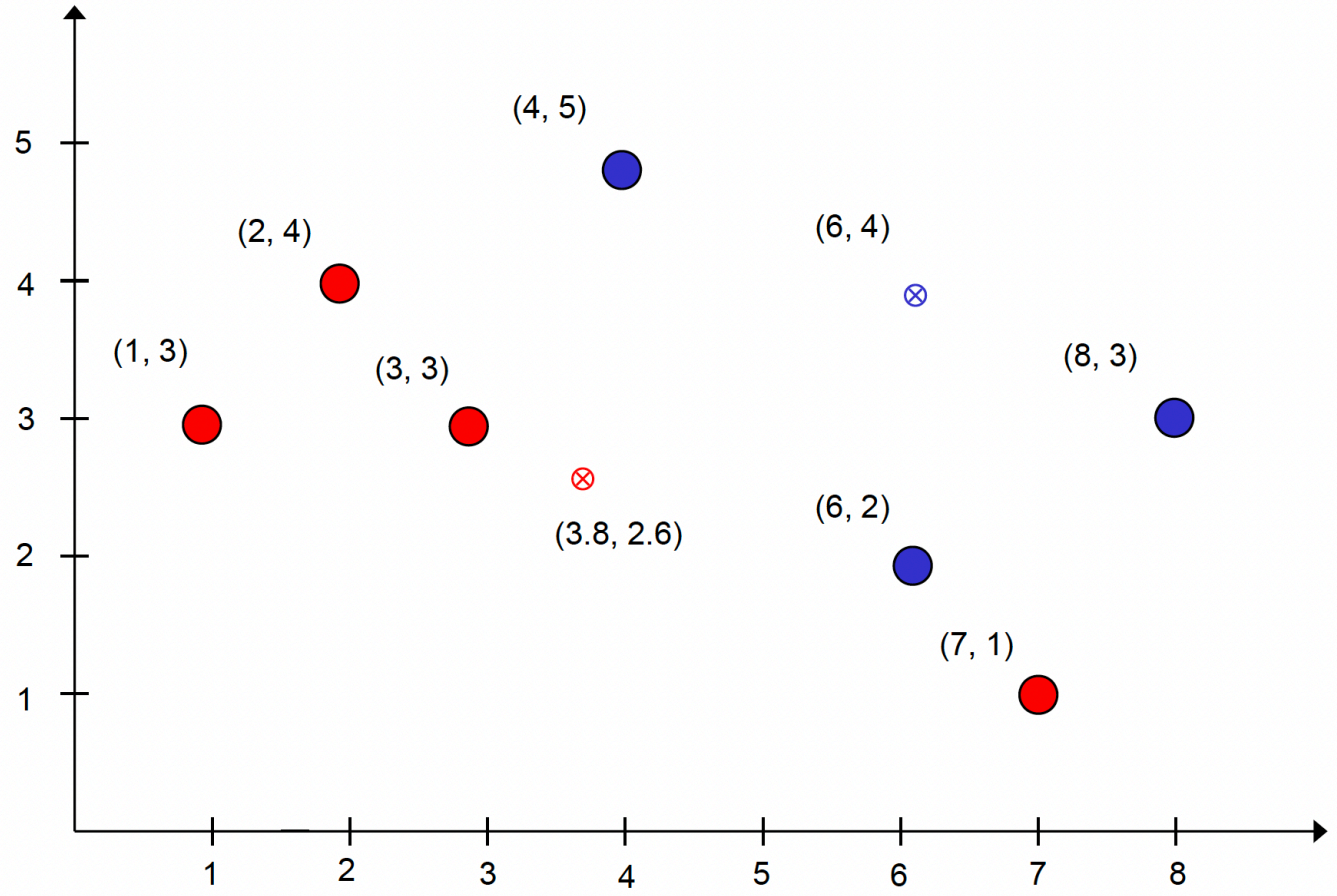
repeat distance calculation.



The Greedy Method: Clustering

K-means: Example

Now: calculate all other distances...



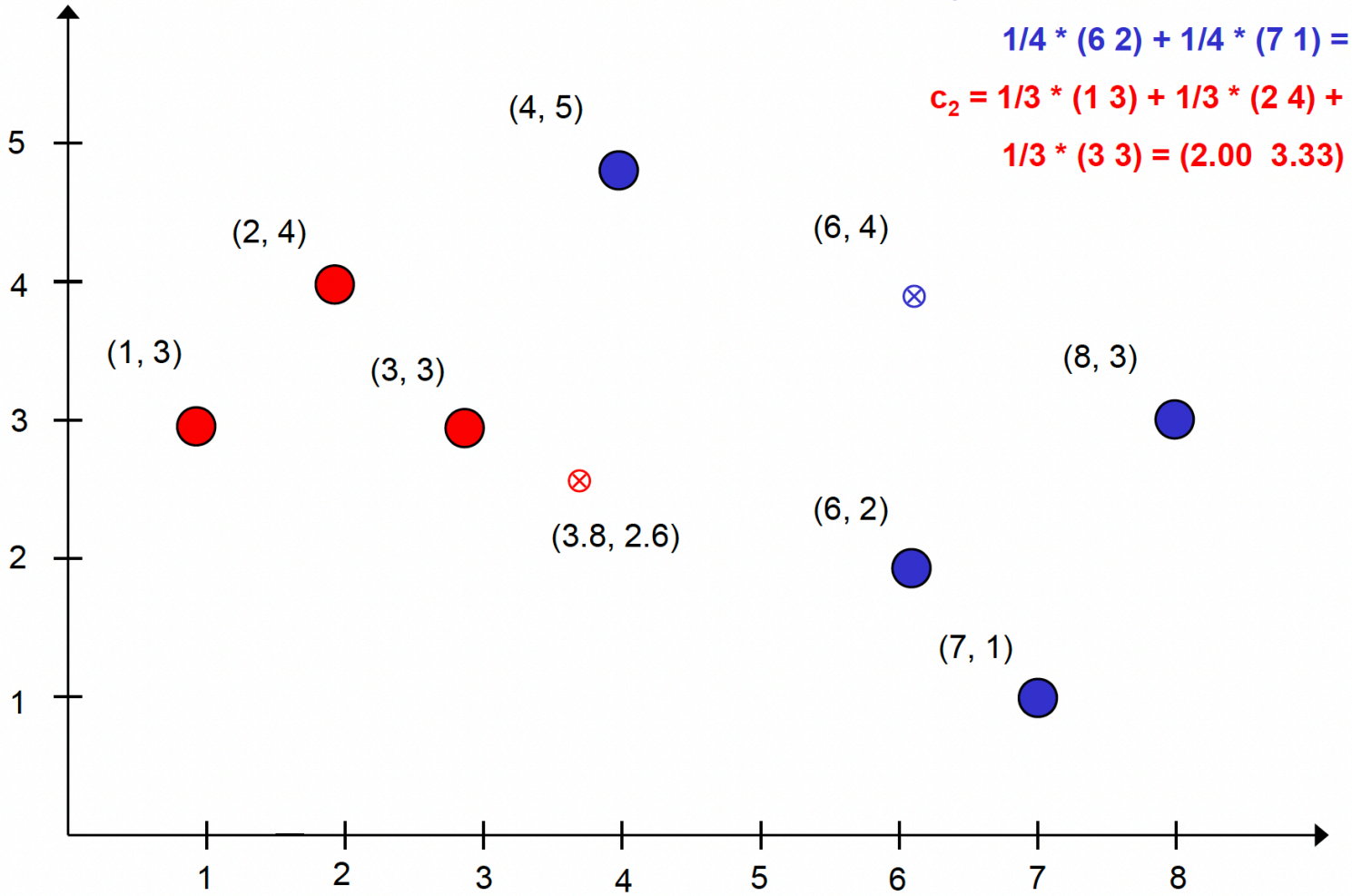
The Greedy Method: Clustering

K-means: Example

Now: move cluster centers to be the average of data points.

c_1 = 1/4 * (4 5) + 1/4 * (8 3) +
1/4 * (6 2) + 1/4 * (7 1) = (5.67 2.67)

c_2 = 1/3 * (1 3) + 1/3 * (2 4) +
1/3 * (3 3) = (2.00 3.33)

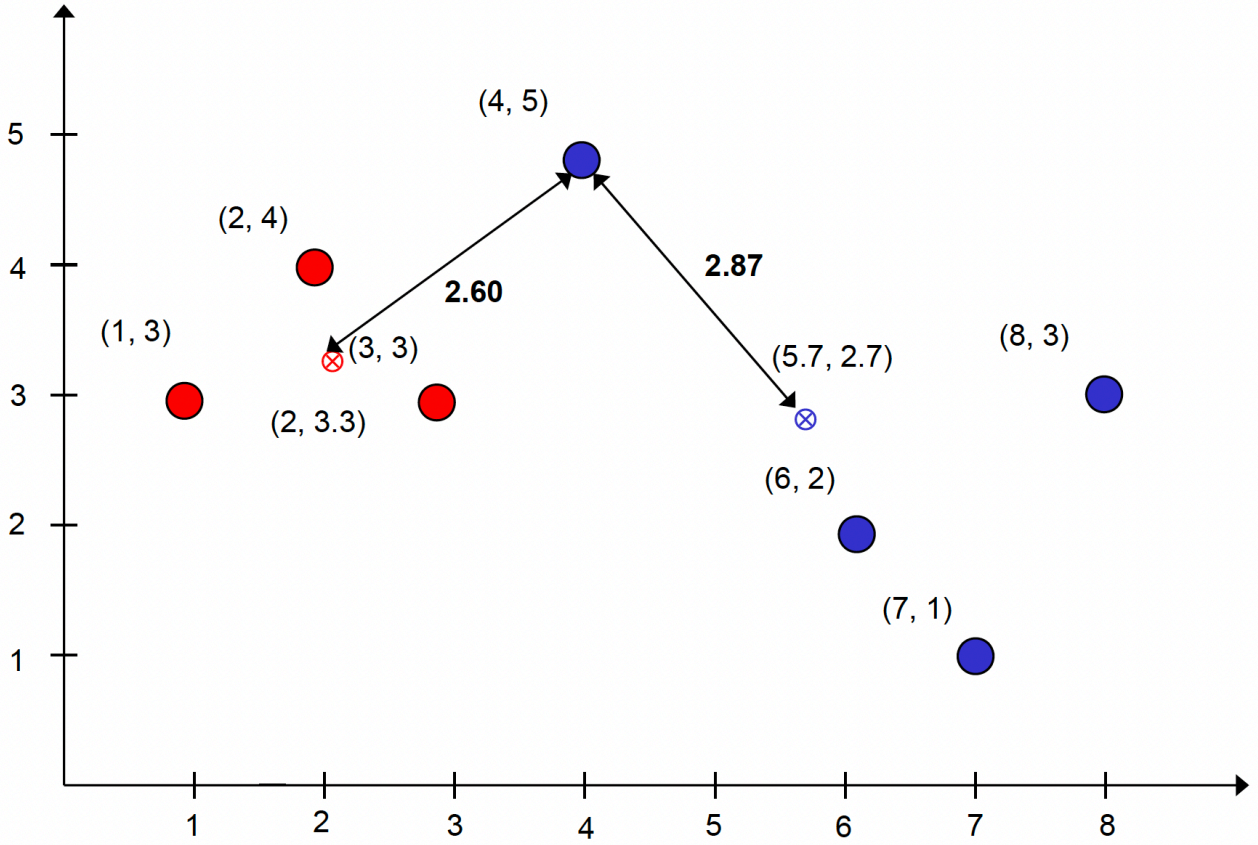


The Greedy Method: Clustering

K-means: Example

Now: start next iteration

calculate distances again...



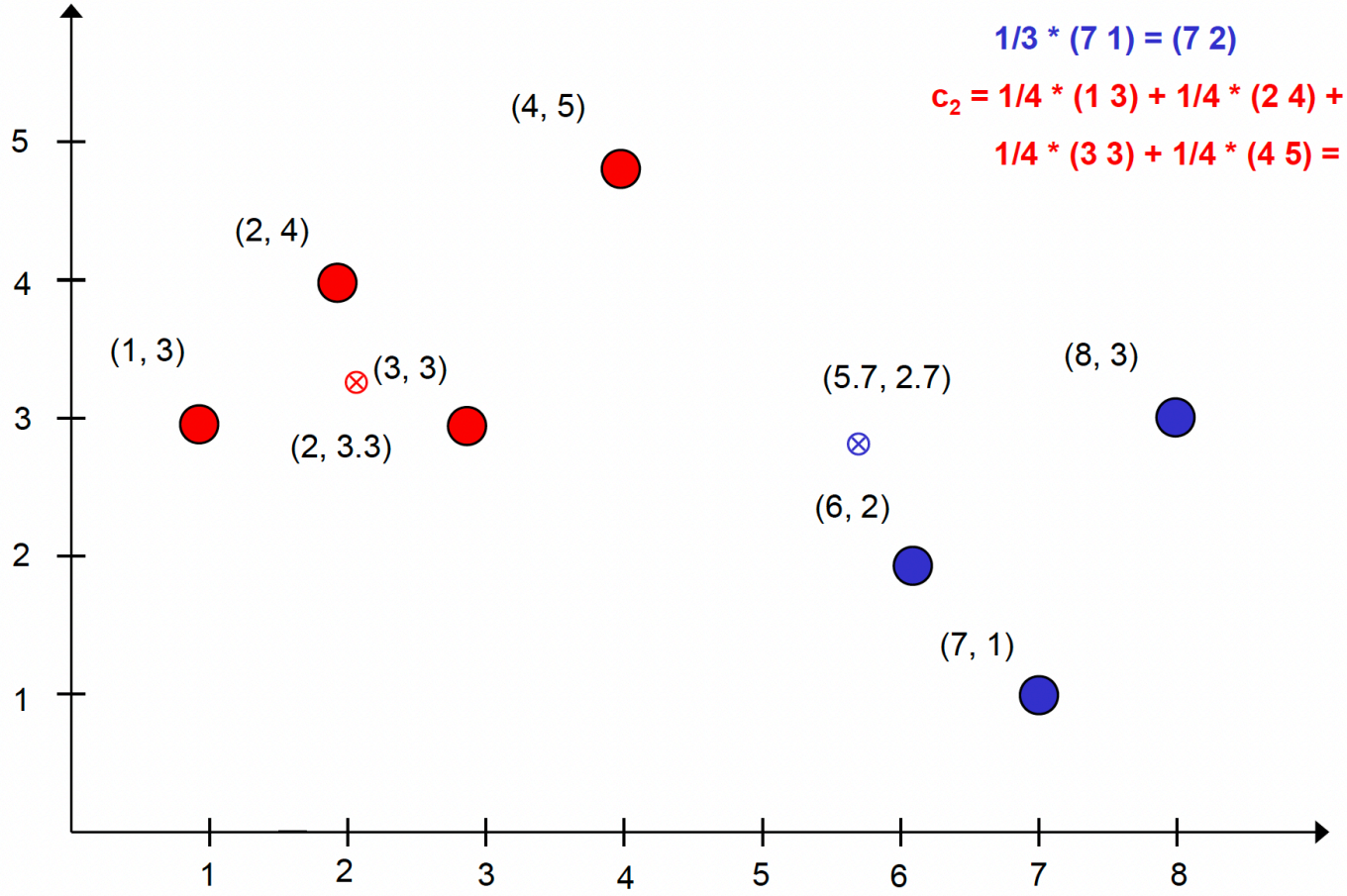
The Greedy Method: Clustering

K-means: Example

Now: move cluster centers to be the average of data points.

c_1 = 1/3 * (6 2) + 1/3 * (8 3) +
1/3 * (7 1) = (7 2)

c_2 = 1/4 * (1 3) + 1/4 * (2 4) +
1/4 * (3 3) + 1/4 * (4 5) = (2.5 3.75)

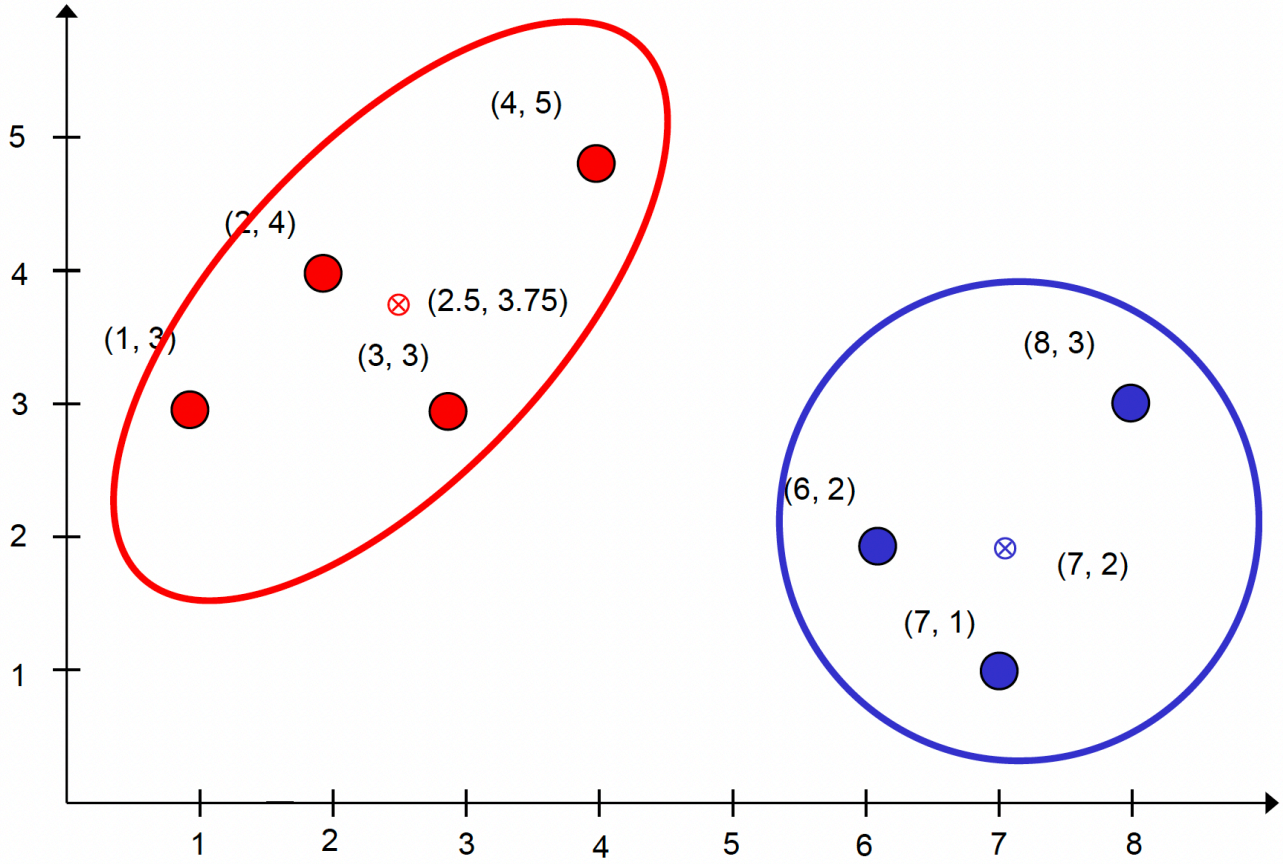


The Greedy Method: Clustering

K-means: Example

Now: if we calculate all distances, no data points will change color.

This means, we can stop!



Expectation-Maximization (EM)

1. An iterative method to find MLE (or MAP) estimates of some latent variables in statistical models where model depends on these parameters.

2. Expectation-maximization algorithm as a clustering algorithm (EM-T): Each cluster can be characterized as a probability distribution and the goal is to iteratively find the MLE of unknown parameters of clusters.

   Video: https://www.youtube.com/watch?v=REypj2sy_5U

The Greedy Method: Expectation-maximization algorithm for clustering

- **Expectation-maximization algorithm for clustering**
  - Data is continually and indiscriminately visited until convergence
  - Sensitive to initialization, Greedy algorithm, Local optimum problem
  - Becomes overwhelmed as data grows $\Rightarrow$ slow convergence

The Greedy Method: Expectation-maximization algorithm for clustering

- The algorithm iteratively alternates between:
  1. Each piece of data is compared and assigned to each Gaussian distribution with some likelihood
  2. Each Gaussian distribution is altered to better represent the data assigned to it during the previous phase .
- Iteration continues until the set of means is stable:

$$\textbf{until} \quad \sum_{i=1}^{k} \|\mu_i^t - \mu_i^{t-1}\|_2^2 \leq \epsilon$$

The Greedy Method: Expectation-maximization algorithm for clustering

- **General Run-time:** $O(ik(d^3 + nd^2))$

  [E-Step]

  invert Σ; compute $|\Sigma_i|$; $O(d^3) \overset{k \text{ clusters}}{\rightarrow} O(kd^3)$

  evaluate density; $O(d^2) \overset{k,n \text{ clusters, points}}{\rightarrow} O(knd^2)$

  [M-Step]

  update Σ; $\overset{k \text{ clusters}}{\rightarrow} O(knd^2)$

- **Best run-time:** $O(inkd)$

  $O(knd)$ for the E-Step and $O(knd)$ for the M-Step

The Greedy Method: Expectation-maximization algorithm for clustering

EXPECTATION-MAXIMIZATION $(\mathbf{D}, k, \epsilon)$:

1 $t \leftarrow 0$

// Initialization

2 Randomly initialize $\boldsymbol{\mu}_1^t, \ldots, \boldsymbol{\mu}_k^t$

3 $\boldsymbol{\Sigma}_i^t \leftarrow \mathbf{I},\ \forall i = 1, \ldots, k$

4 $P^t(C_i) \leftarrow \frac{1}{k},\ \forall i = 1, \ldots, k$

5 **repeat**

6 $\quad t \leftarrow t + 1$

$\quad$ // Expectation Step

7 $\quad$ **for** $i = 1, \ldots, k$ *and* $j = 1, \ldots, n$ **do**

8 $\quad\quad w_{ij} \leftarrow \frac{f(\mathbf{x}_j | \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \cdot P(C_i)}{\sum_{a=1}^{k} f(\mathbf{x}_j | \boldsymbol{\mu}_a, \boldsymbol{\Sigma}_a) \cdot P(C_a)}$ // posterior probability $P^t(C_i | \mathbf{x}_j)$

$\quad$ // Maximization Step

9 $\quad$ **for** $i = 1, \ldots, k$ **do**

10 $\quad\quad \boldsymbol{\mu}_i^t \leftarrow \frac{\sum_{j=1}^{n} w_{ij} \cdot \mathbf{x}_j}{\sum_{j=1}^{n} w_{ij}}$ // re-estimate mean

11 $\quad\quad \boldsymbol{\Sigma}_i^t \leftarrow \frac{\sum_{j=1}^{n} w_{ij} (\mathbf{x}_j - \boldsymbol{\mu}_i)(\mathbf{x}_j - \boldsymbol{\mu}_i)^T}{\sum_{j=1}^{n} w_{ij}}$ // re-estimate covariance matrix

12 $\quad\quad P^t(C_i) \leftarrow \frac{\sum_{j=1}^{n} w_{ij}}{n}$ // re-estimate priors

13 **until** $\sum_{i=1}^{k} \left\| \boldsymbol{\mu}_i^t - \boldsymbol{\mu}_i^{t-1} \right\|^2 \leq \epsilon$

References

1. Hetland, M. L. (2014). Python Algorithms: mastering basic algorithms in the Python Language. Apress.

2. Lee, K. D., Lee, K. D., & Steve Hubbard, S. H. (2015). *Data Structures and Algorithms with Python*. Springer.

3.Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python*. Hoboken: Wiley

4.Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.

5.Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-wesley professional.

6.Tan, P. N., Steinbach, M., & Kumar, V. (2013). Data mining cluster analysis: basic concepts and algorithms. *Introduction to data mining*, 487-533.