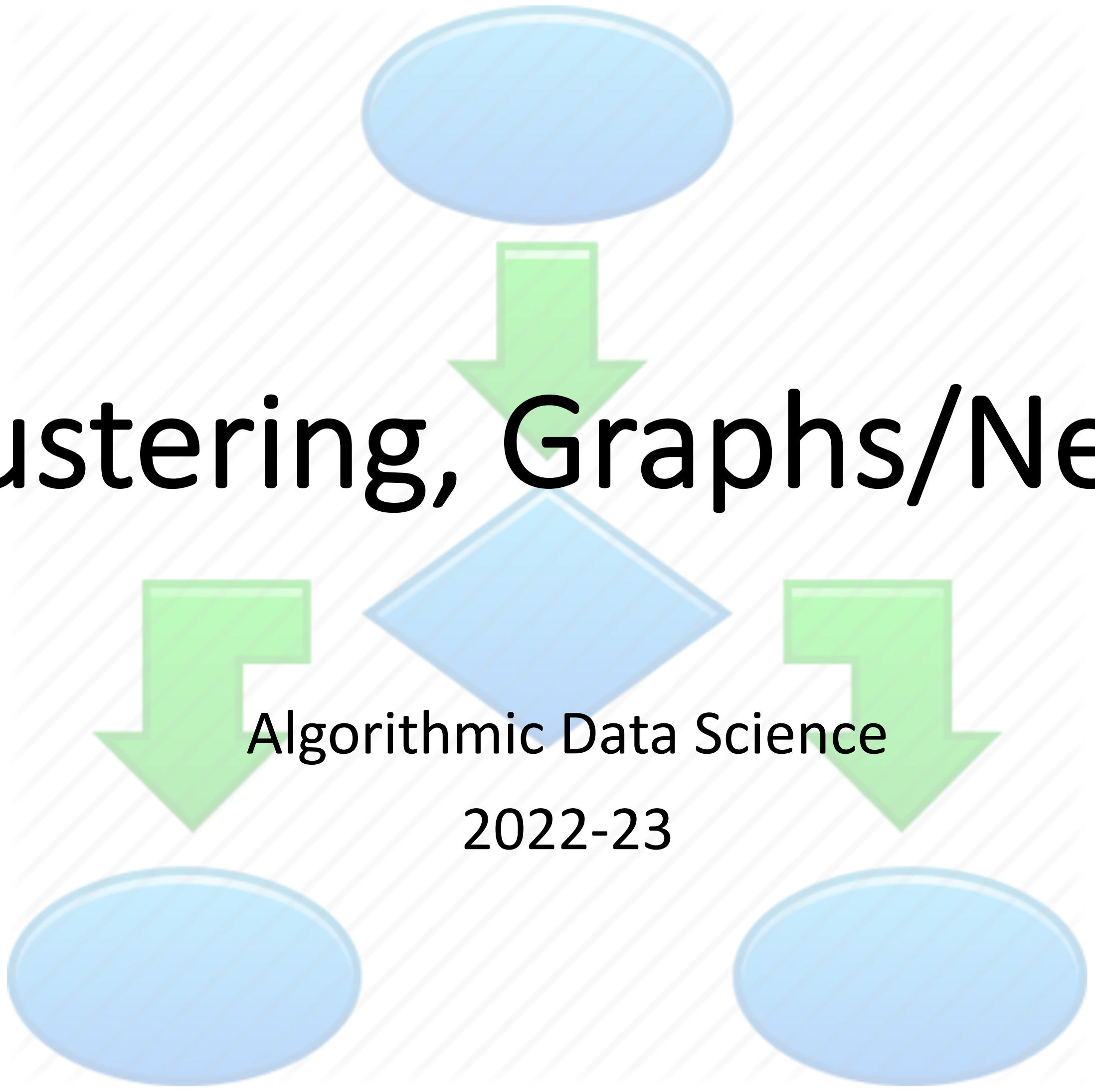


Week 8: Clustering, Graphs/Networks 1

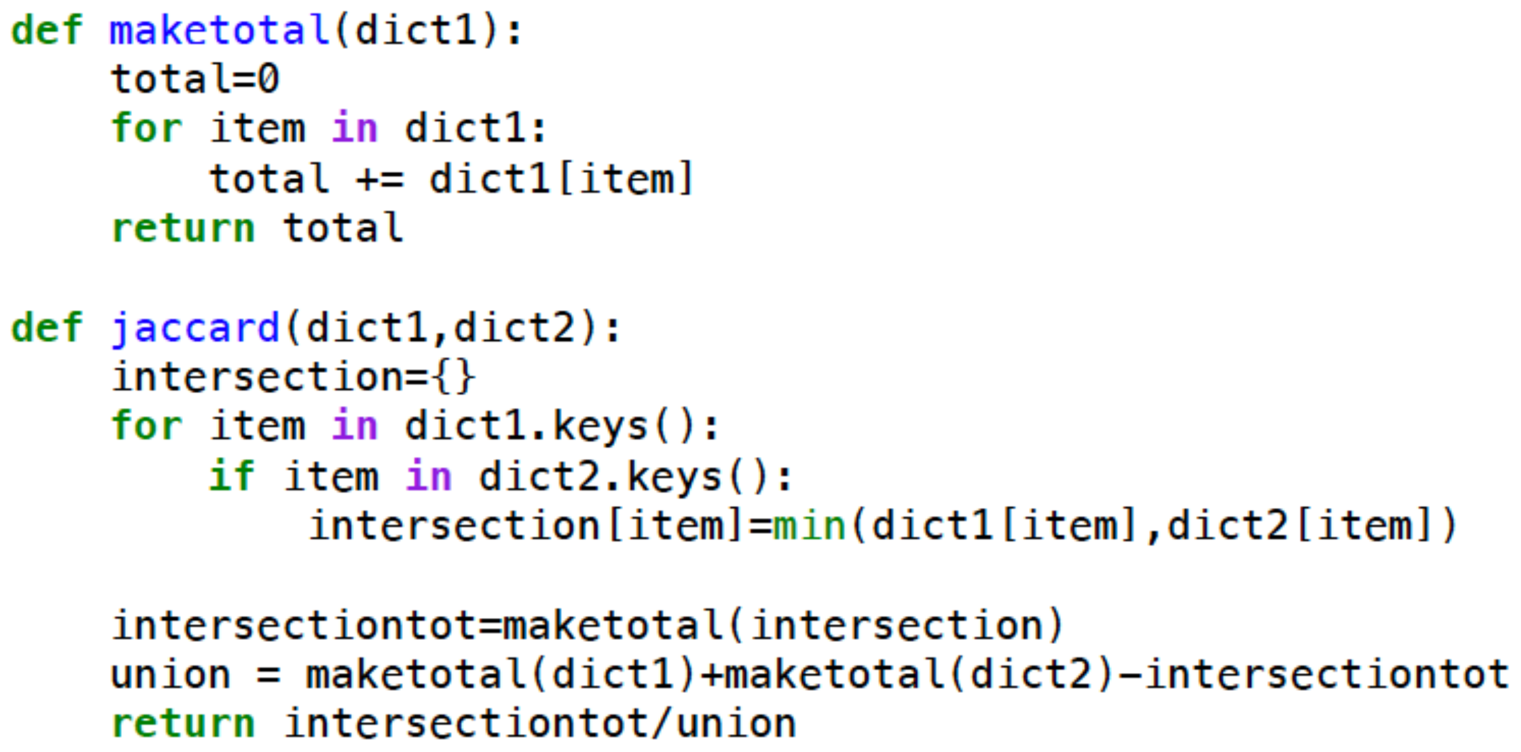
Algorithmic Data Science

2022-23

Dr Adam Barrett

Warm up

Recall the code for the computing the Jaccard similarity of 2 documents represented as bags of words:



```
def maketotal(dict1):
    total=0
    for item in dict1:
        total += dict1[item]
    return total

def jaccard(dict1,dict2):
    intersection={}
    for item in dict1.keys():
        if item in dict2.keys():
            intersection[item]=min(dict1[item],dict2[item])

    intersectiontot=maketotal(intersection)
    union = maketotal(dict1)+maketotal(dict2)-intersectiontot
    return intersectiontot/union
```

What is the space complexity of this algorithm in O notation?

How much extra memory is required, beyond the memory taken up by the inputs, in O notation?

Week	Who	Topic
1	Barrett	Data structures and data formats
2	Barrett	Algorithmic complexity. Sorting.
3	Barrett	Matrices: Manipulation and computation
4	Barrett	Similarity analysis
5	Rosas	Processes and concurrency
6	Rosas	Distributed computation
7	Barrett	Map/reduce
8	**Barrett**	**Clustering, graphs/networks**
9	Barrett	Graphs/networks, PageRank algorithm
10	Barrett	Databases
11		*independent study*

Overview

- Clustering.

- **Graphs / networks I**

  - Breadth-first and depth-first searching

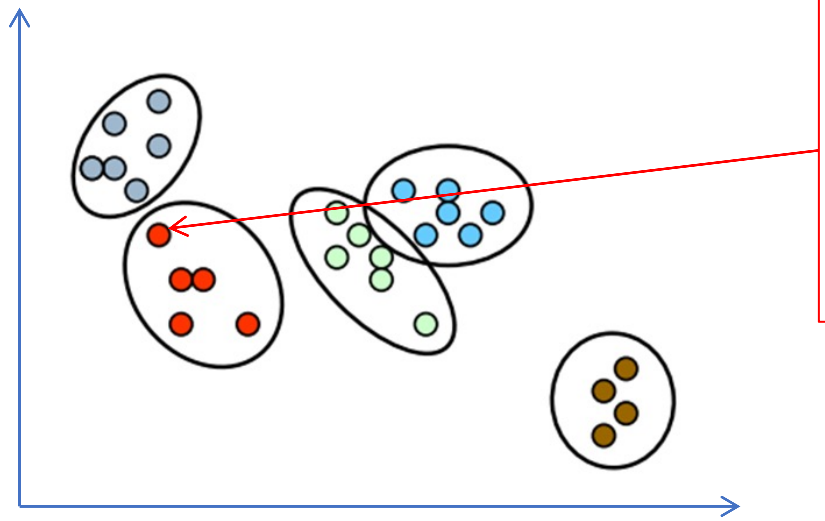
  - Dijkstra’s algorithm (introduction)

Next week:

- Dijkstra’s algorithm (example)
- Mining social media networks

Clustering

Clustering (or cluster analysis) is the task of grouping objects in such a way that objects in the same group (called a **cluster**) are **more similar** to each other than to those in other clusters.

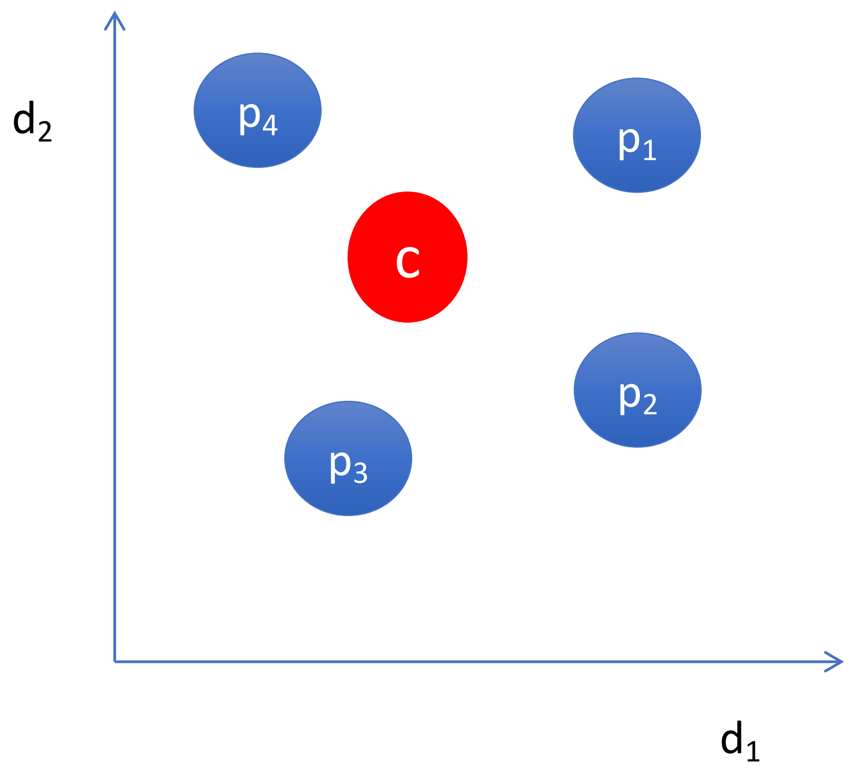


In general, there is no one "correct" clustering. For example, a different algorithm may assign the red point closest to the grey cluster to the grey cluster.

Clustering algorithms are all affected by choice of similarity / distance measure and scaling of dimensions

K-means

- Simple partitioning approach based on the idea of centroids or prototypes

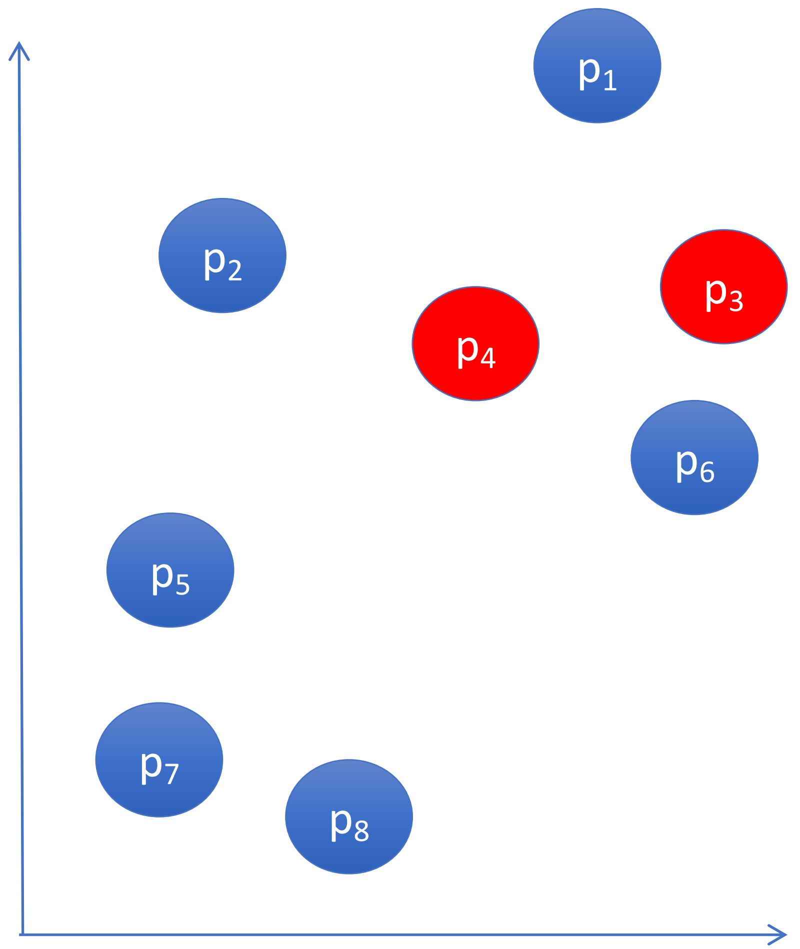


The centroid (middle) of any group of points in a Euclidean space can be found by taking the mean on every dimension.

$$c_{d_j} = \frac{\sum_{i=0}^{n} p_{d_j,i}}{n}$$

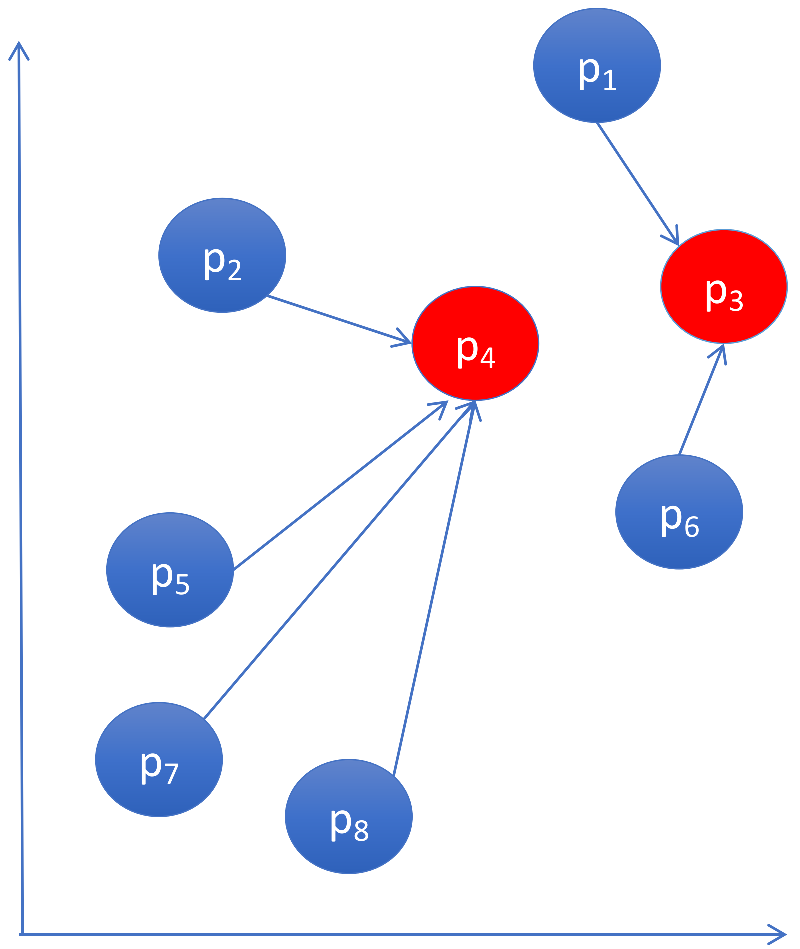
K-means

```
1. Select K points
   as initial
   centroids
2. while centroids
   changing:
    1. Form K clusters
       by assigning each
       point to its
       nearest centroid
    2. Recompute
       centroid of the
       cluster
```



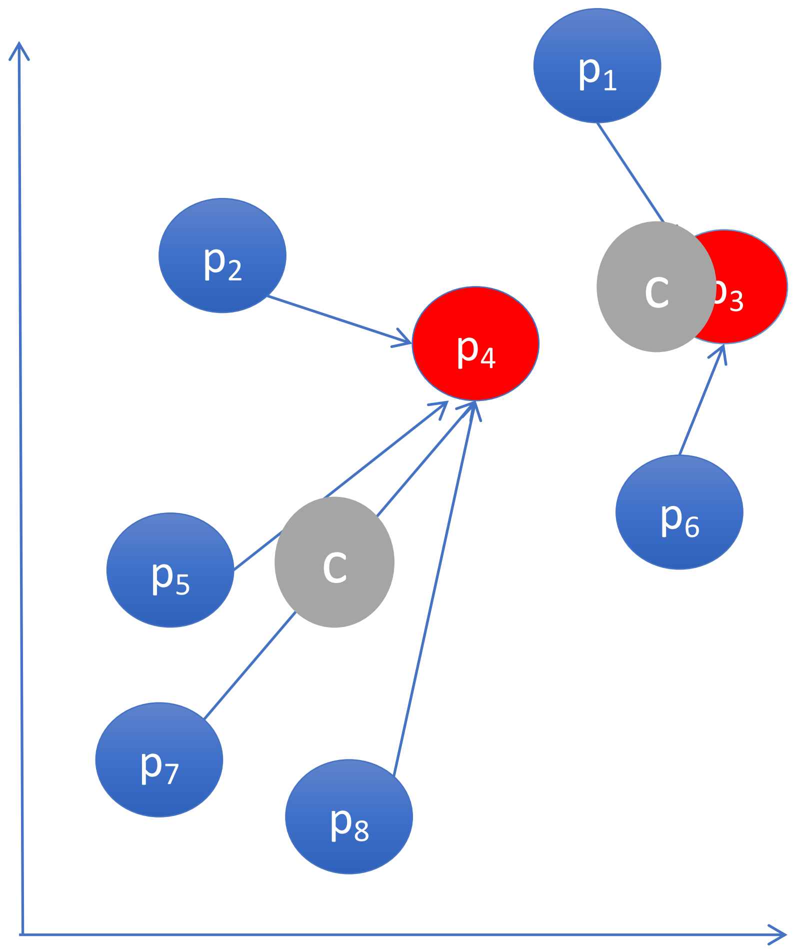
K-means

```
1. Select K points
   as initial
   centroids
2. while centroids
   changing:
     1. Form K clusters
        by assigning each
        point to its
        nearest centroid
     2. Recompute
        centroid of the
        cluster
```



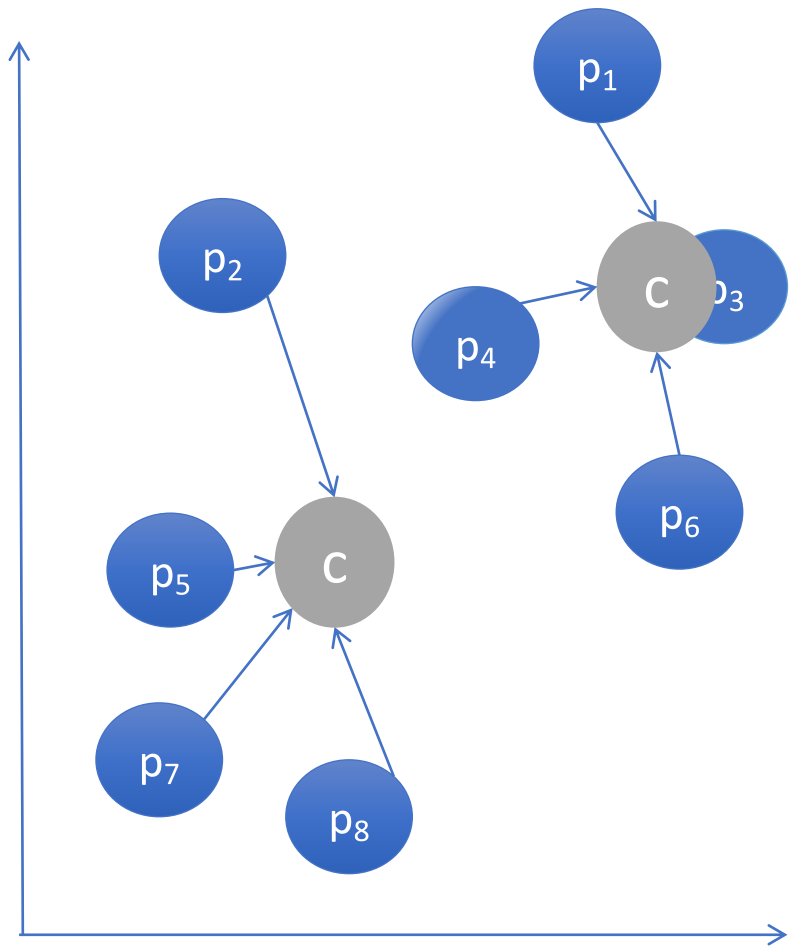
K-means

```
1. Select K points
   as initial
   centroids
2. while centroids
   changing:
    1. Form K clusters
       by assigning each
       point to its
       nearest centroid
    2. Recompute
       centroid of the
       cluster
```



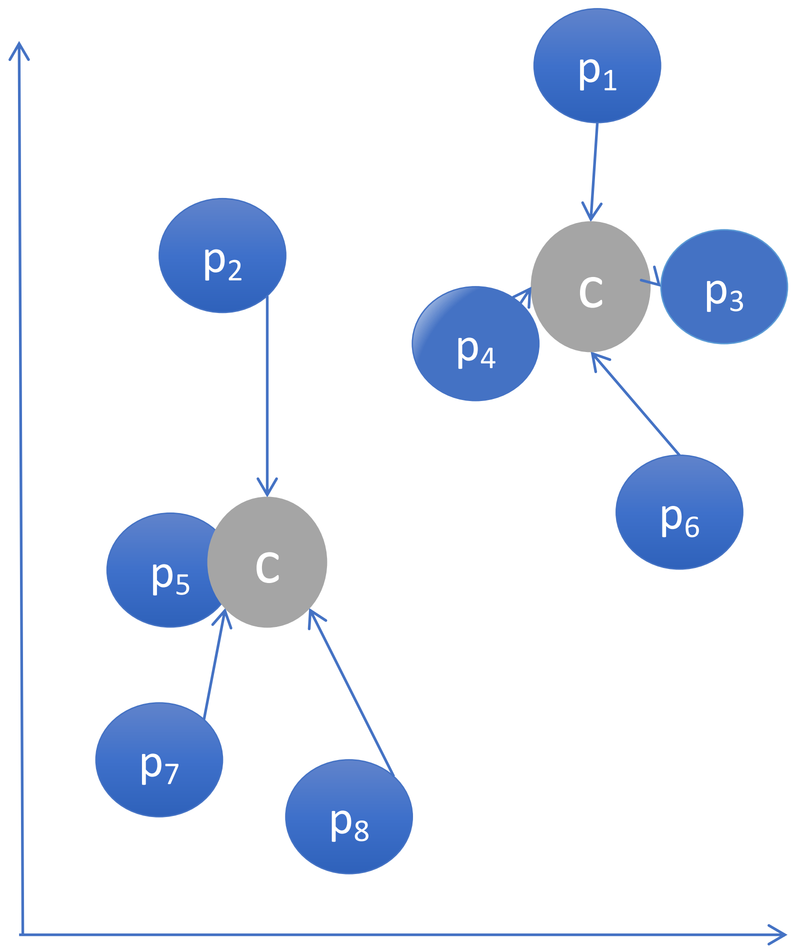
K-means

```
1. Select K points
   as initial
   centroids
2. while centroids
   changing:
    1. Form K clusters
       by assigning each
       point to its
       nearest centroid
    2. Recompute
       centroid of the
       cluster
```



K-means

```
1. Select K points
   as initial
   centroids
2. while centroids
   changing:
    1. Form K clusters
       by assigning each
       point to its
       nearest centroid
    2. Recompute
       centroid of the
       cluster
```



Complexity of K-Means

```
1. Select K points as initial centroids
2. while centroids changing:
    1. Form K clusters by assigning each point to its nearest centroid
    2. Recompute centroid of the cluster
```

Run-time = O(IKnd)

where:

I = number of iterations

K = number of clusters

n = number of data points

d = number of dimensions

Which of these factors is likely to dominate?

Disadvantages of K-Means

- Need to know the number of clusters in advance
- Each point is only assigned to a single cluster (hard clustering)
- Flat structure (no clusters within clusters)
- May converge on local mimimum i.e., suboptimal clustering (this can be overcome to some extent by repeated random initialisations)

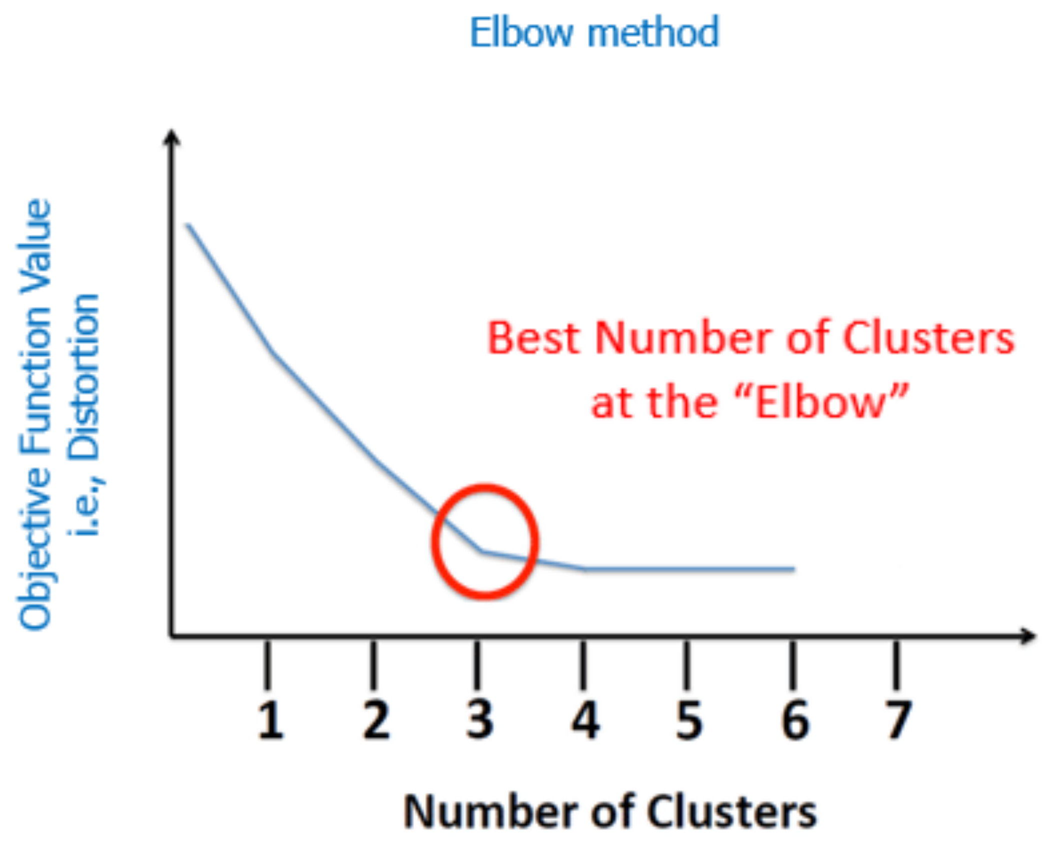
How many clusters?

- Minimize sum of distances to centroids? Called distortion or inertia

Distortion $=\sum_{i=1}^{N} ||x_i - C_i||^2$

- If you have N clusters, every point is at the centroid of its own cluster containing just itself.

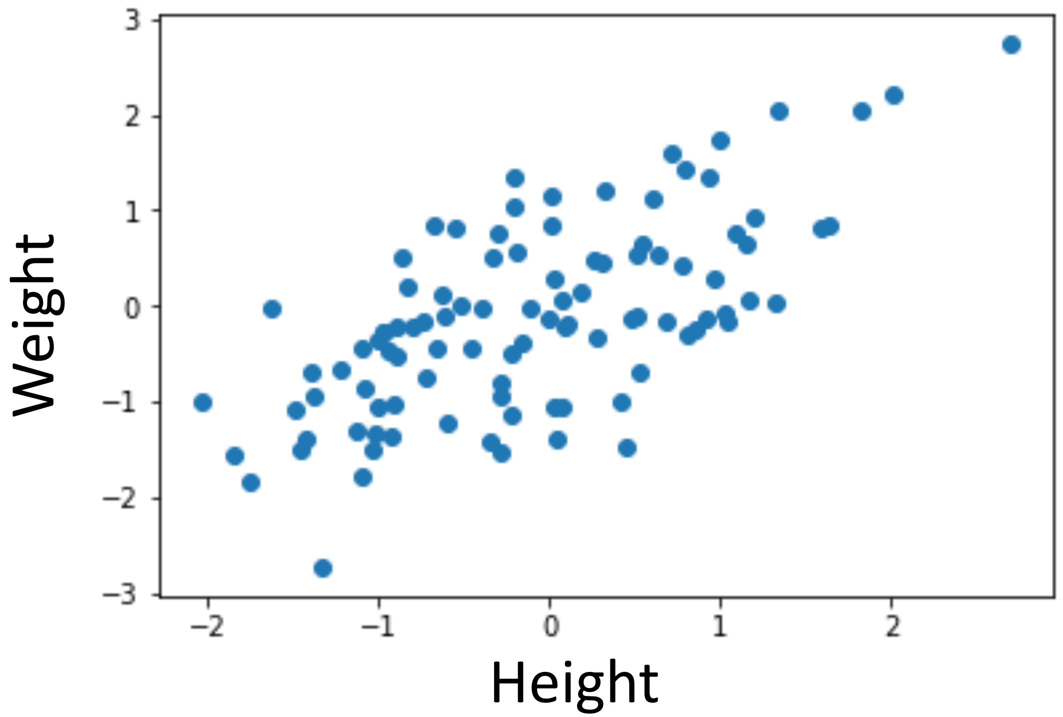
- Looking for substantial gain in having more clusters



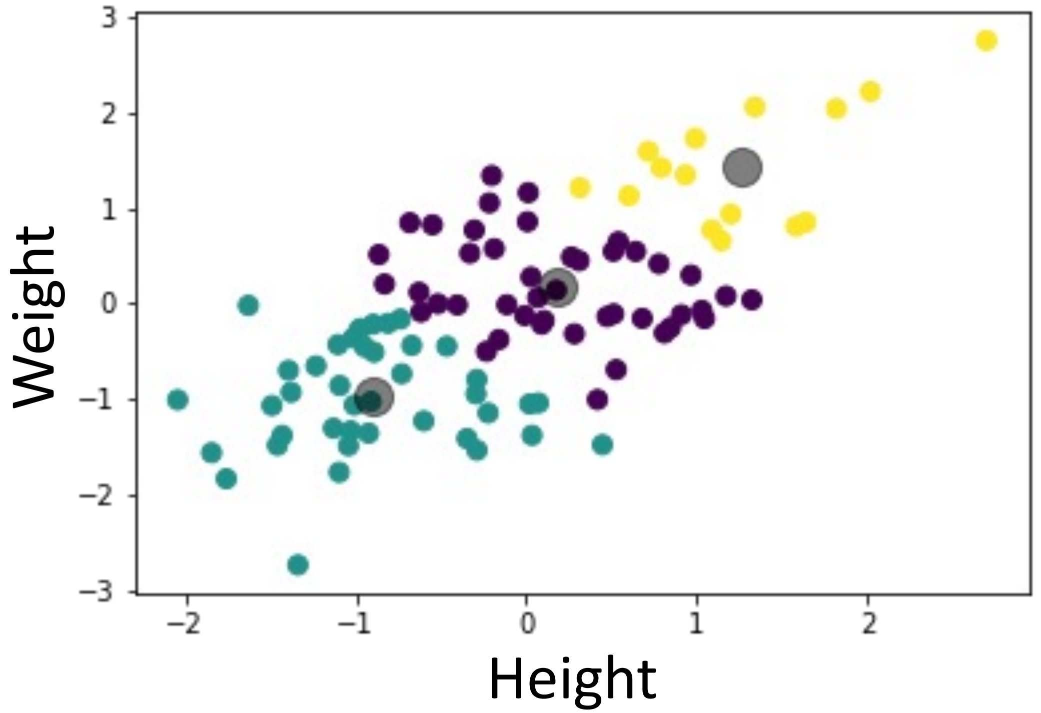
- E.g. Data points are accident prone areas, centroids are where to put Hospital Emergency Units.

Example T-shirts

- Suppose I am a manufacturer of T-shirts and I want to make 3 different sizes. How do I optimise the sizes?



Example T-shirts



But is this optimal?

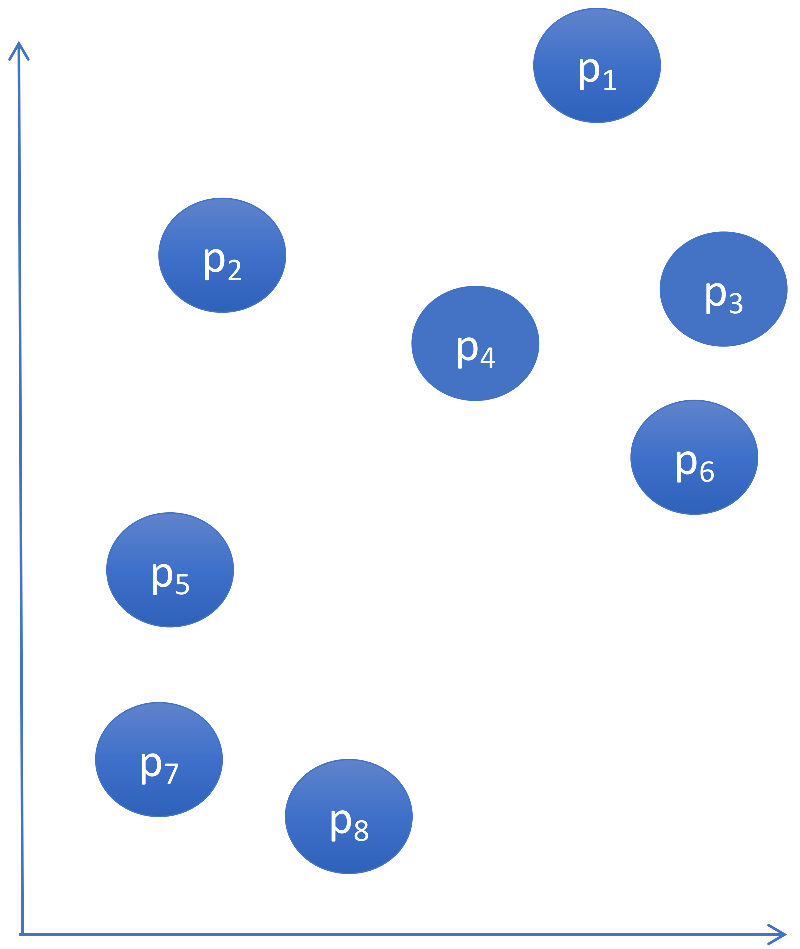
Have to consider many factors.

Agglomerative hierarchical clustering

- is an agglomerative technique which builds up clusters by repeatedly merging the closest pair of clusters
- Do not need to know the number of clusters in advance
- Hierarchical (clusters have internal structure)

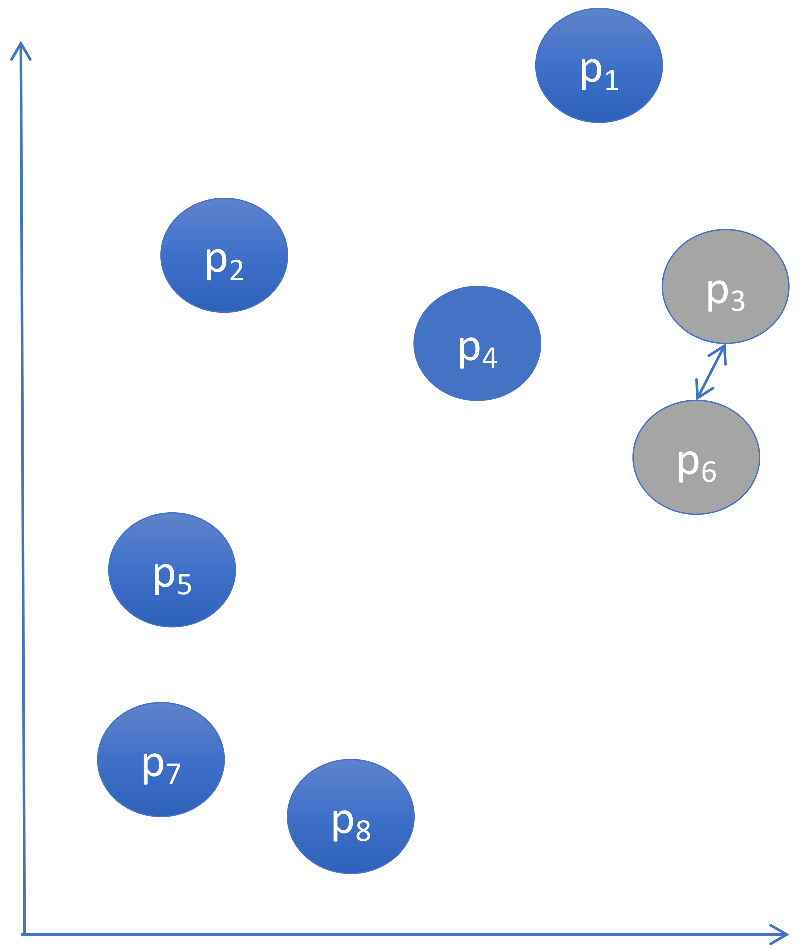
Agglomerative hierarchical clustering

```
1. Initialise n clusters as the n data points
2. Find closest pair <$p_i$,$p_j$> of clusters with distance d
3. while d < threshold:
    1. Merge clusters <$p_i$,$p_j$>
    2. Find closest pair <$p_i$, $p_j$> with distance d
```



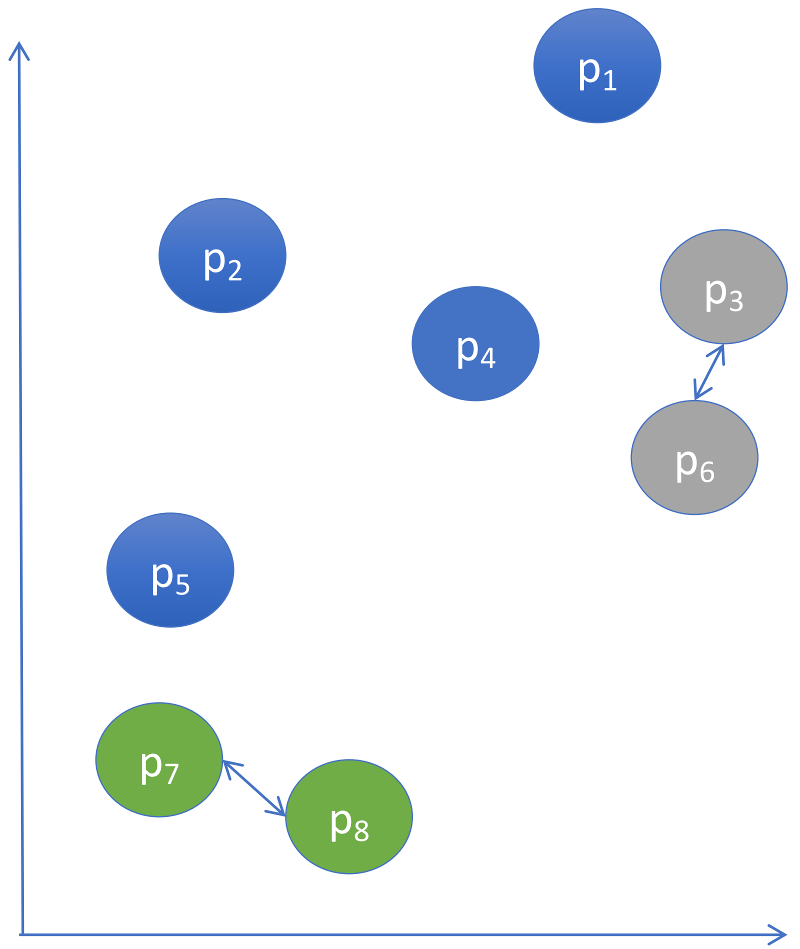
Agglomerative hierarchical clustering

```
1. Initialise n clusters as the n data points
2. Find closest pair <$p_i$,$p_j$>  of clusters with distance d
3. while d < threshold:
    1. Merge clusters <$p_i$,$p_j$>
    2. Find closest pair <$p_i$, $p_j$> with distance d
```



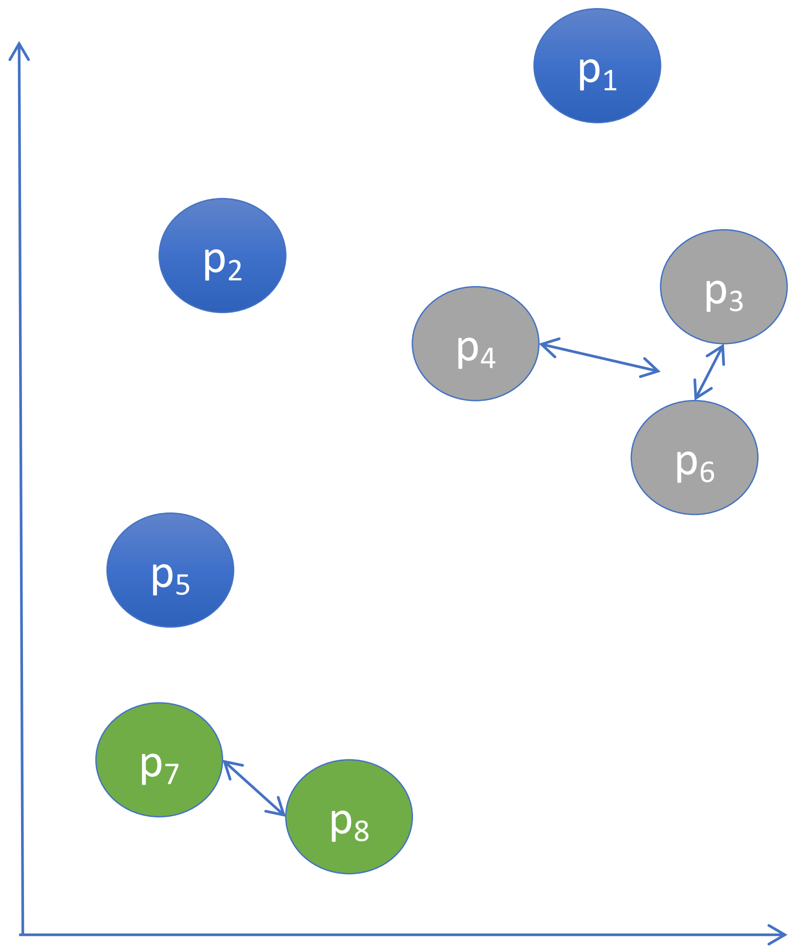
Agglomerative hierarchical clustering

```
1. Initialise n clusters as the n data points
2. Find closest pair $<p_i,p_j>$ of clusters with distance d
3. while d < threshold:
    1. Merge clusters $<p_i,p_j>$
    2. Find closest pair $<p_i, p_j>$ with distance d
```



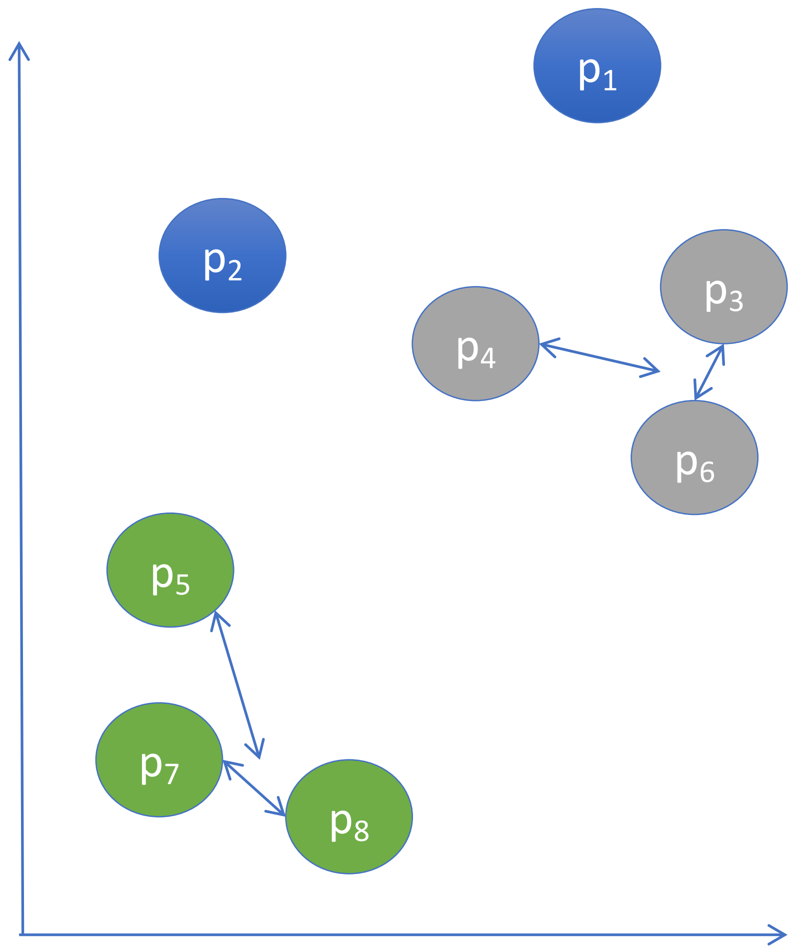
Agglomerative hierarchical clustering

```
1. Initialise n clusters as the n data points
2. Find closest pair <$p_i$,$p_j$>  of clusters with distance d
3. while d < threshold:
    1. Merge clusters <$p_i$,$p_j$>
    2. Find closest pair <$p_i$, $p_j$> with distance d
```



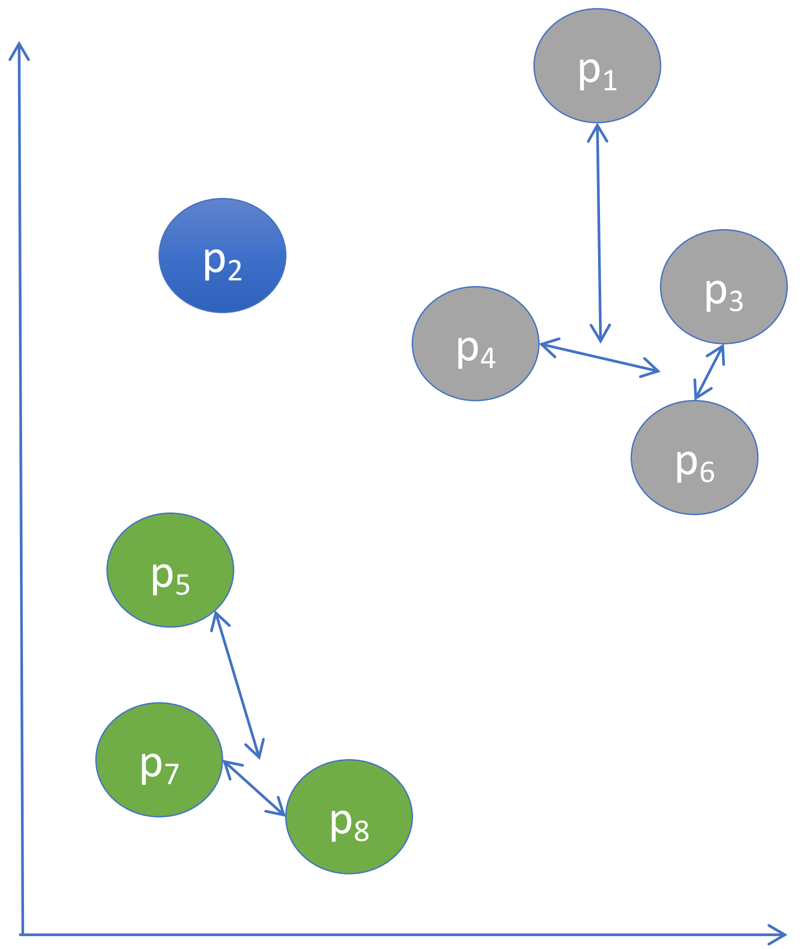
Agglomerative hierarchical clustering

```
1. Initialise n clusters as the n data points
2. Find closest pair <$p_i$,$p_j$> of clusters with distance d
3. while d < threshold:
    1. Merge clusters <$p_i$,$p_j$>
    2. Find closest pair <$p_i$, $p_j$> with distance d
```



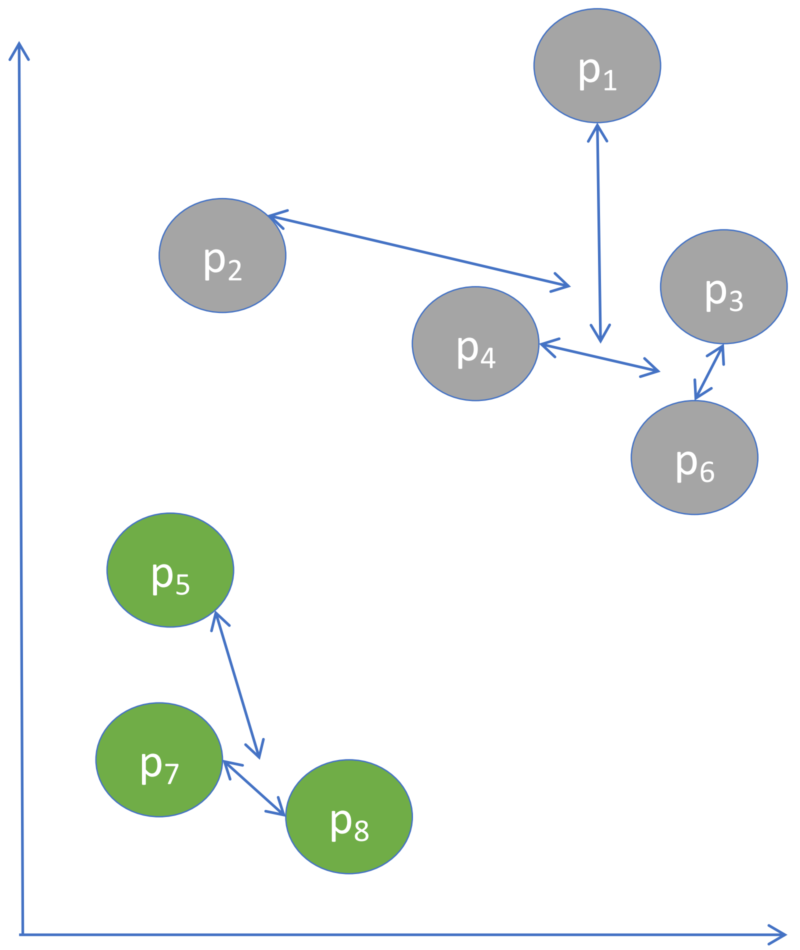
Agglomerative hierarchical clustering

```
1. Initialise n clusters as the n data points
2. Find closest pair <$p_i$,$p_j$> of clusters with distance d
3. while d < threshold:
    1. Merge clusters <$p_i$,$p_j$>
    2. Find closest pair <$p_i$, $p_j$> with distance d
```



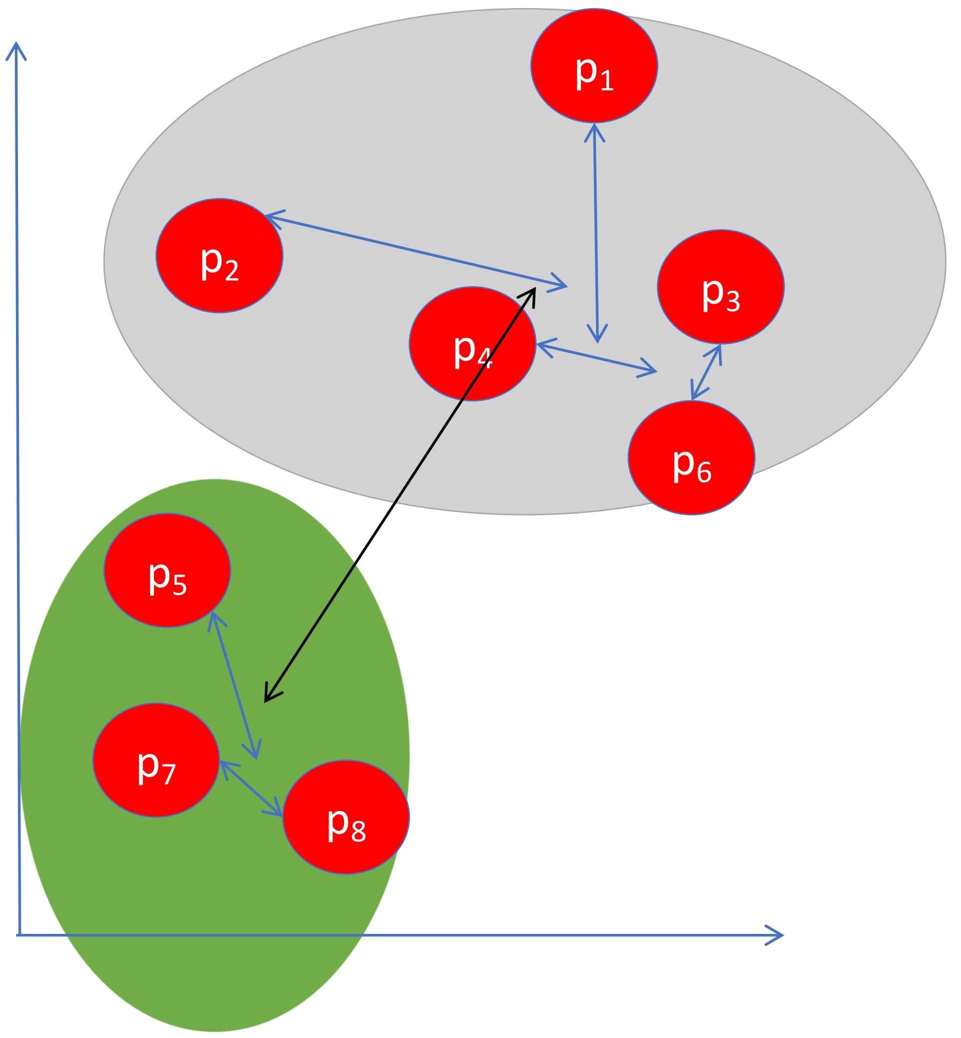
Agglomerative hierarchical clustering

```
1. Initialise n clusters as the n data points
2. Find closest pair <$p_i$,$p_j$> of clusters with distance d
3. while d < threshold:
    1. Merge clusters <$p_i$,$p_j$>
    2. Find closest pair <$p_i$, $p_j$> with distance d
```



Agglomerative hierarchical clustering

```
1. Initialise n clusters as the n data points
2. Find closest pair <$p_i$,$p_j$> of clusters with distance d
3. while d < threshold:
    1. Merge clusters <$p_i$,$p_j$>
    2. Find closest pair <$p_i$, $p_j$> with distance d
```



Complexity of agglomerative hierarchical clustering

- Computing all pairs similarities is $O(n^2)$
- Sorting n^2 similarity scores is $O(n^2 \log n^2) = O(n^2 \log n)$
- After a merge, the similarity scores can be updated in $O(n)$
- Number of clusters reduces by 1 each time so there are up to n-1 merges
- runtime = $O(n^2 \log n)$ where n is the number of data points
- For a large number of features, all pairs similarity is $O(dn^2)$ which can be rather big.

Question to ponder

If you have *n* bags (where *n* is large e.g., *n = 10,000*), which is computationally more expensive?

a. Finding the *c* nearest neighbours of each bag (where *c* is much less than *n* e.g., *c = 100*)?
b. Organising the bags into k clusters, where k is not too big?

Some answer on next 2 slides

Analysis of K-Nearest Neighbours

- Finding similarity of two objects is O(d) where d is the number of dimensions
  - Can be much less if sparse
- All-pairs similarity is $O(n^2)$. But ….
  - Using LSH we can massively reduce the constant
- Finding top-1 is O(n). So finding top-c is O(cn)
- Overall, dominant term $O(n^2)$

Complexity of K-Means

```
1. Select K points
   as initial
   centroids
2. while centroids
   changing:
    1. Form K clusters
       by assigning each
       point to its
       nearest centroid
    2. Recompute
       centroid of the
       cluster
```

Run-time = O(IKnd)

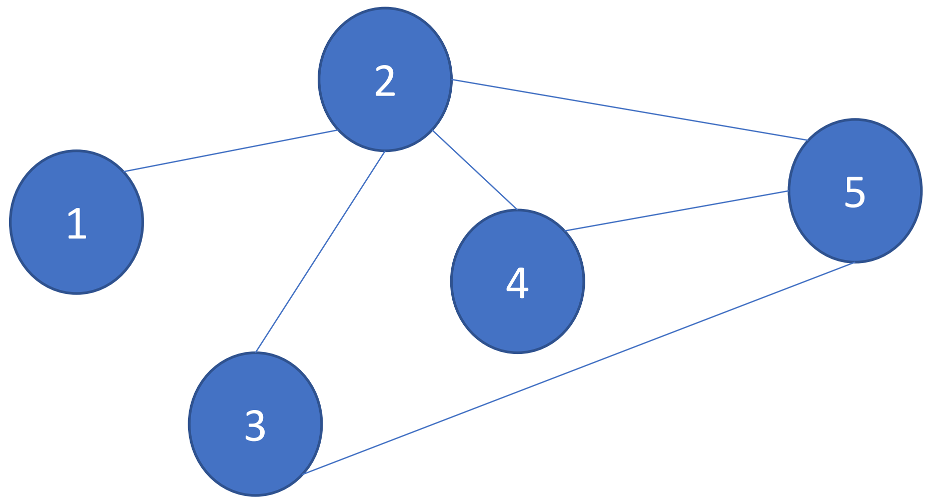
where:
I = number of iterations
K = number of clusters
n = number of data points
d = number of dimensions

Which of these factors is likely to dominate?

This is O(n).

Graphs / Networks

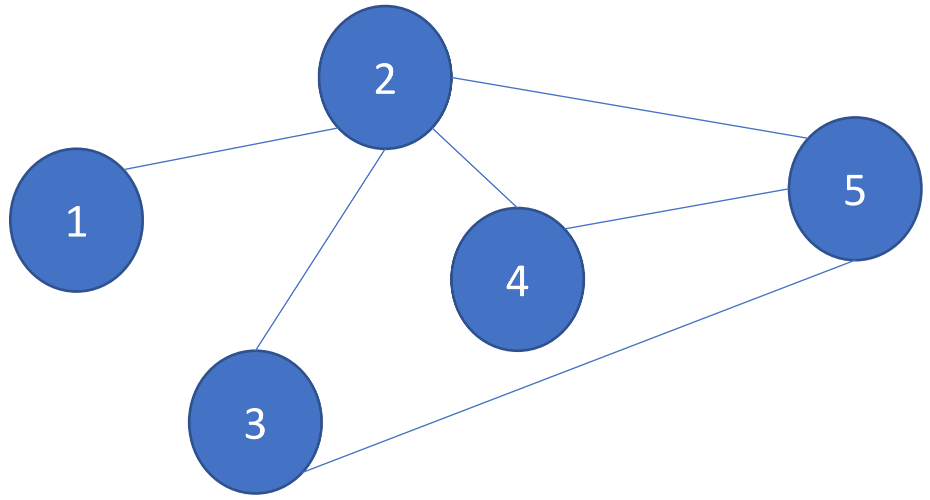
- Informally, a graph is a set of objects and a set of connections between them



- More formally,
  - the objects are referred to as **vertices** (or nodes).
  - the connections are referred to as **edges** (or arcs).
  - A graph G is the pair (V,E) where V is a finite set of vertices and E is a binary relation on V.

Undirected graphs

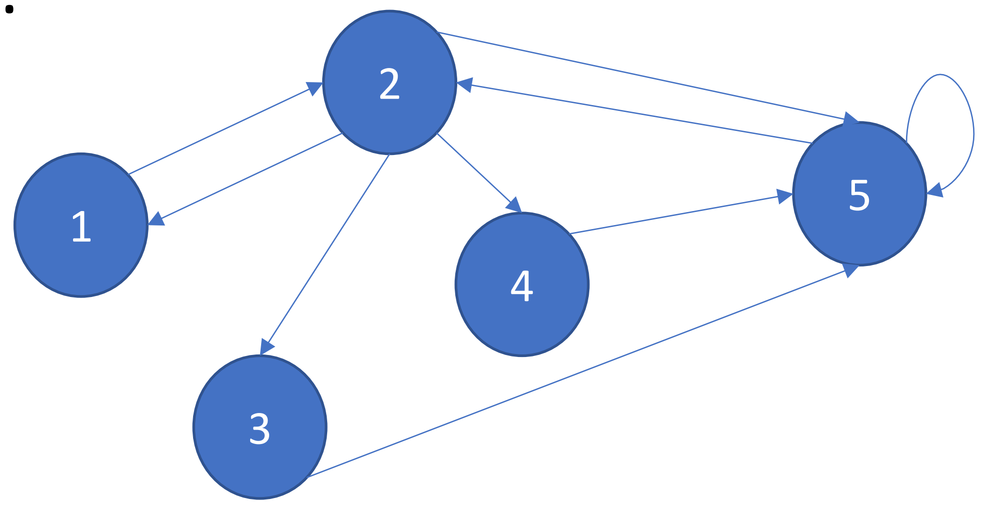
- If the relationship between the objects is symmetric then the graph is **undirected**.



- In this example, the vertices might represent people on a social networking site such as Facebook
- The edges represent friendships between them

Directed graphs

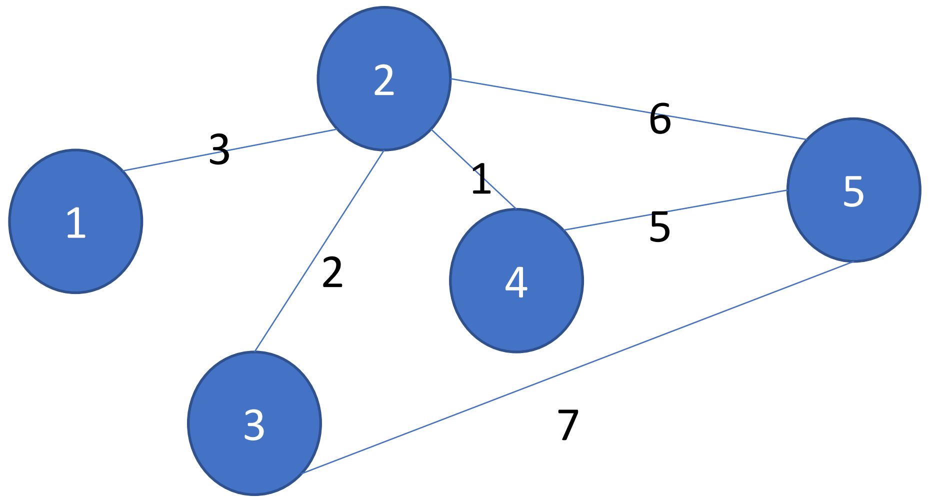
- If the relationship between the objects is asymmetric then the graph is **directed**.



- In this example, the vertices might represent pages on the Internet.
- The edges might represent links between them

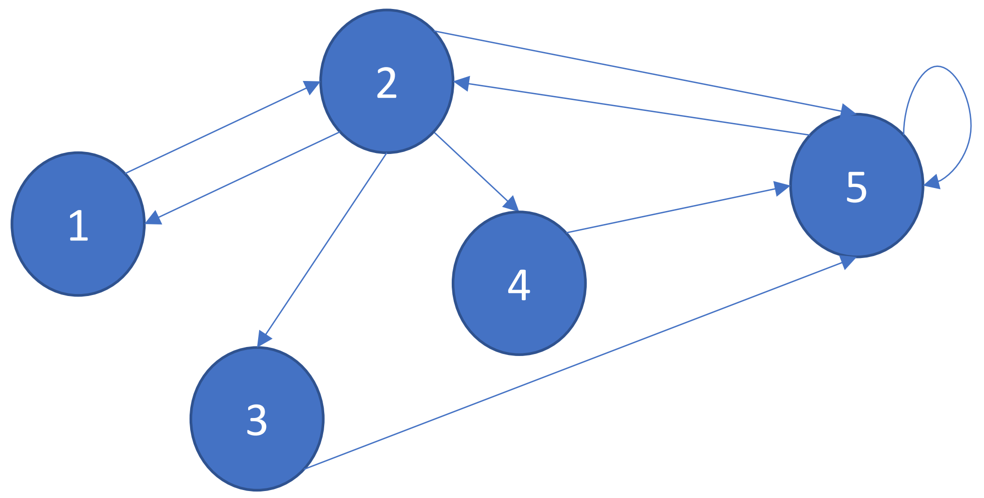
Weighted graphs

- Both directed and undirected graphs can be weighted i.e., have a weight associated with each of the edges.



- In this example, the vertices might represent physical locations
- The edges represent roads between the locations.
- The weights represent the distance or cost of travelling along those roads

More terminology

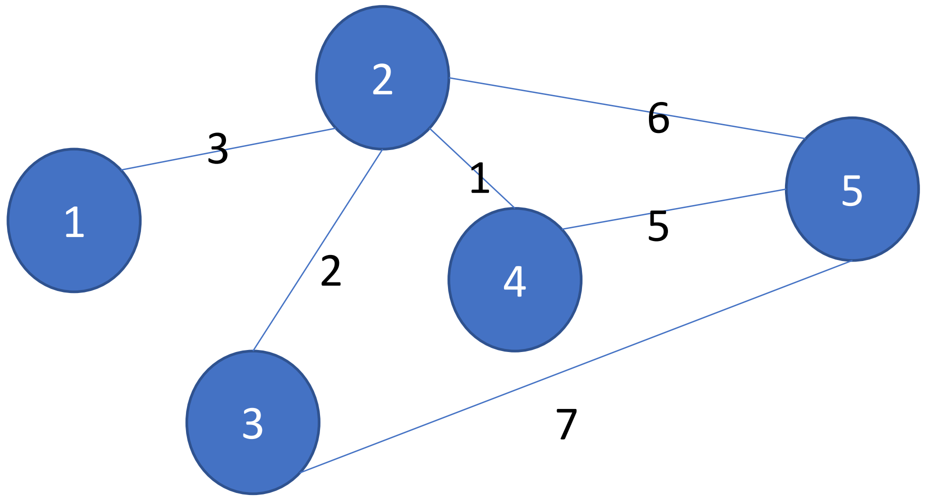


- If (u,v) is an edge in the graph, we say v is **adjacent** to u.
- The **out-degree** of a vertex is the number of edges leaving it
- The **in-degree** of a vertex is the number of edges entering it

- A **path** from a vertex v_0 to a vertex v_k is a sequence of vertices $<v_0,v_1,\ldots,v_k>$ such that $(v_{i-1},v_i) \in E$ for i=1,2,...k
- The **length** of a path is the number of edges in the path.
- If there is a path from u to v then we say that v is **reachable** from u
- A **cycle** is a path from a vertex to itself.
- An undirected graph is **connected** if every pair of vertices is connected by a path.
- A directed graph is **strongly connected** if every two vertices are reachable from each other

Is the graph shown strongly connected?

Representing graphs



Adjacency list representation:
An array, list or dictionary of lists which record, for each vertex, its adjacent vertices (with weights)

{1:[(2,3)],
2: [(1,3),(3,2),(5,6),(4,1)],
3:[(2,2),(5,7)],
4:[(5,5),(2,1)],
5:[(3,7),(2,6),(4,5)]

memory requirements = O(V+E)

Adjacency (or characteristic) matrix representation:
A matrix where each rows and columns refers to vertices and the value of m_{ij} is the weight on the edge from vertex i to vertex j (0 or null if no edge)

	1	**2**	**3**	**4**	**5**
1	0	3	0	0	0
2	3	0	2	1	6
3	0	2	0	0	7
4	0	1	0	0	5
5	0	6	7	5	0

memory requirements = O(V^2)

Which representation is better?

Searching graphs

Example: A web crawler starts at a given web page, scrapes all of the text from the page, identifies all of the hyperlinks and then visits all of the pages linked to by that page and repeats the process. It stops when it has visited all of pages within a given distance, i.e., number of links, of the original page. How can we be sure that the crawler will visit all of the required pages and terminate?

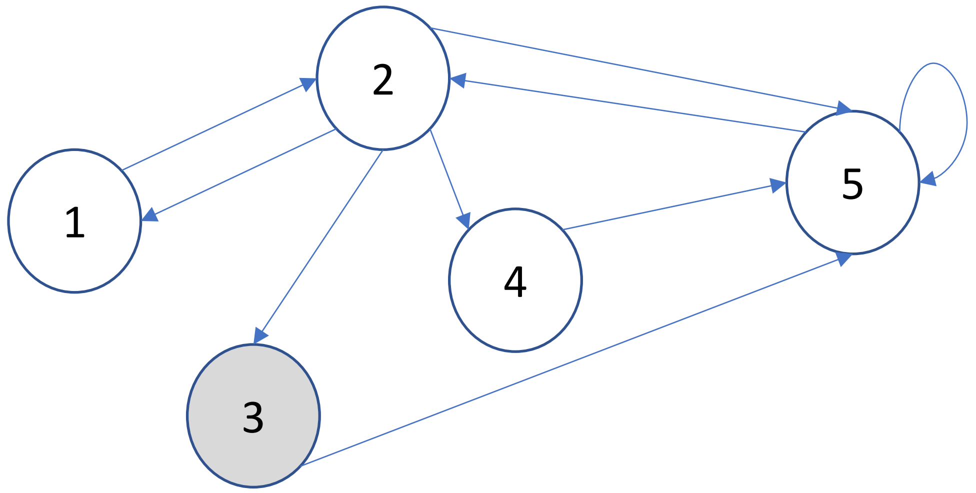
This is a graph problem. Vertices are webpages. Edges are hyperlinks. Edges are directed but unweighted.

Breadth-first search

- Given a graph G = (V,E) and a **source** vertex *s*, breadth-first search systematically explores the edges of G to discover every vertex that is reachable from s
- It computes the distance (fewest number of edges) from *s* to all such reachable vertices
- It produces a **breadth-first tree** with root *s* that contains all such reachable vertices
- It is so-named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.

Breadth-first search: initialisation

```
BFS(G=(V,E),s):
for each vertex u ∈ V − {s}:
   colour[u]=white
   d[u] = ∞
   π[u] = Null
colour[s] = grey
d[s] = 0
π[s] = Null
Q=[s]
```

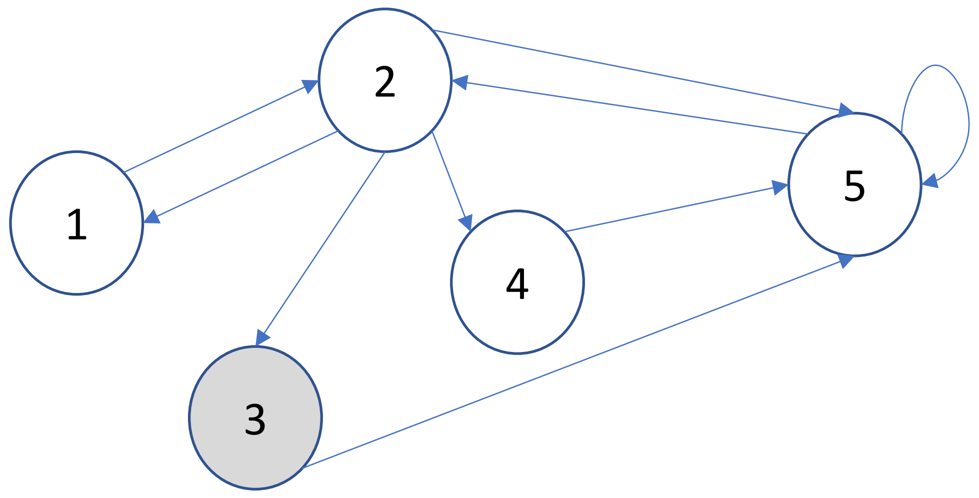


Q=[3]

Adj=	colour=	d =	π =
{1:[2],	{1:white,	{1:∞,	{1:Null,
2:[1,3,4,5],	2:white,	2:∞,	2:Null,
3:[5],	3:grey,	3:0,	3:Null,
4:[5],	4:white,	4:∞,	4:Null,
5:[2,5]}	5:white}	5:∞}	5:Null}

Breadth-first search: iteration

```
while notempty(Q):
  u = head(Q)
  for each v in Adj(u):
    if colour[v] == white:
      colour[v] = grey
      d[v]=d[u]+1
      π[v]=u
      Q.append(v)
  Q.remove(u)
  colour[u]=black
```

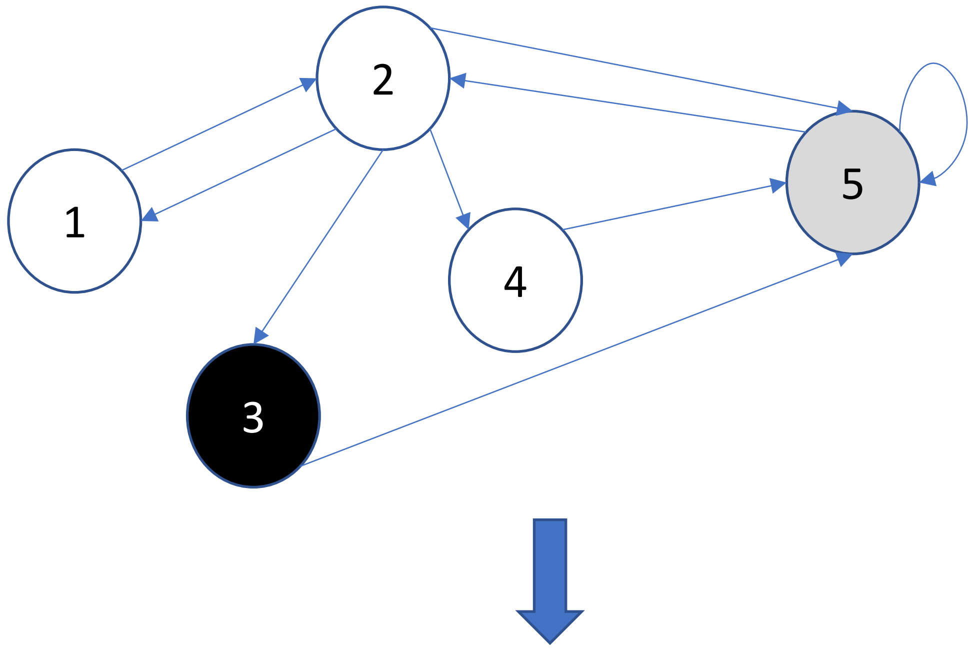


Q=[3]

Adj=	colour=	d =	π =
{1:[2],	{1:white,	{1:∞,	{1:Null,
2:[1,3,4,5],	2:white,	2:∞,	2:Null,
3:[5],	3:grey,	3:0,	3:Null,
4:[5],	4:white,	4:∞,	4:Null,
5:[2,5]}	5:white}	5:∞}	5:Null}

Breadth-first search: iteration

```
while notempty(Q):
  u = head(Q)
  for each v in Adj(u):
    if colour[v] == white:
      colour[v] = grey
      d[v]=d[u]+1
      π[v]=u
      Q.append(v)
  Q.remove(u)
  colour[u]=black
```



Q=[5]

Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

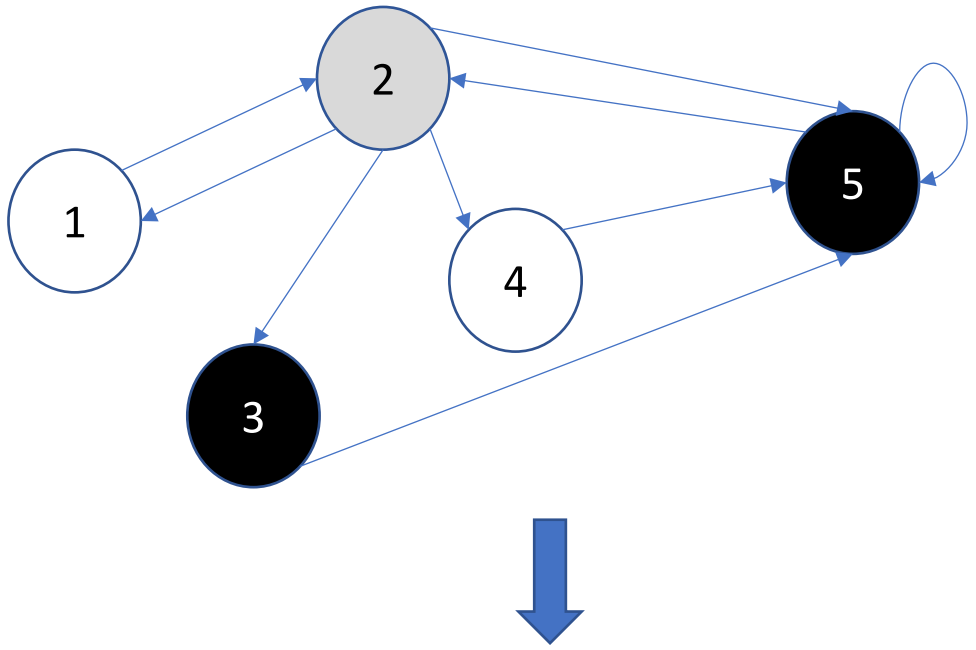
colour=
{1:white,
2:white,
3:black,
4:white,
5:grey}

d =
{1:∞,
2:∞,
3:0,
4:∞,
5:1}

π =
{1:Null,
2:Null,
3:Null,
4:Null,
5:3}

Breadth-first search: iteration

```
while notempty(Q):
  u = head(Q)
  for each v in Adj(u):
    if colour[v] == white:
      colour[v] = grey
      d[v]=d[u]+1
      π[v]=u
      Q.append(v)
  Q.remove(u)
  colour[u]=black
```



Q=[2]

Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

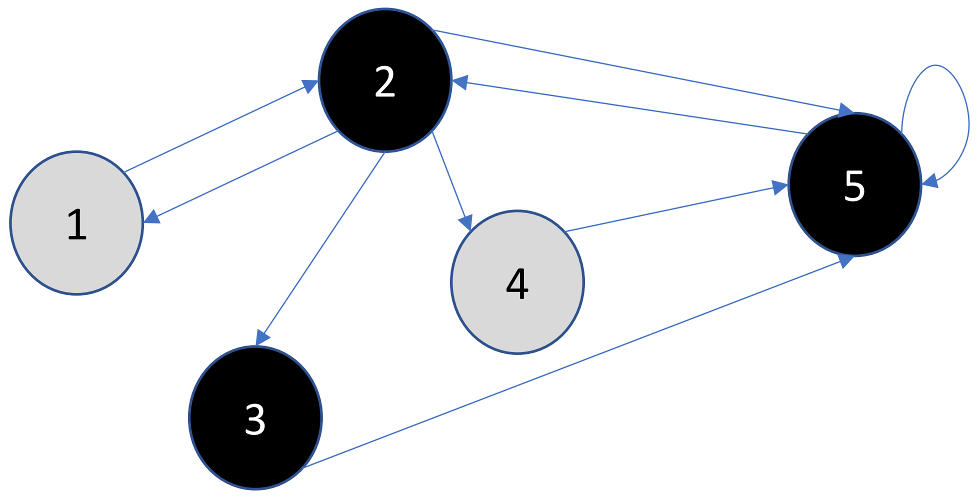
colour=
{1:white,
2:white,
3:black,
4:white,
5:black}

d =
{1:∞,
2:2,
3:0,
4:∞,
5:1}

π =
{1:Null,
2:5,
3:Null,
4:Null,
5:3}

Breadth-first search: iteration

```
while notempty(Q):
  u = head(Q)
  for each v in Adj(u):
    if colour[v] == white:
      colour[v] = grey
      d[v]=d[u]+1
      π[v]=u
      Q.append(v)
  Q.remove(u)
  colour[u]=black
```



Q=[1,4]

Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

colour=
{1:grey,
2:black,
3:black,
4:grey,
5:black}

d =
{1:3,
2:2,
3:0,
4:3,
5:1}

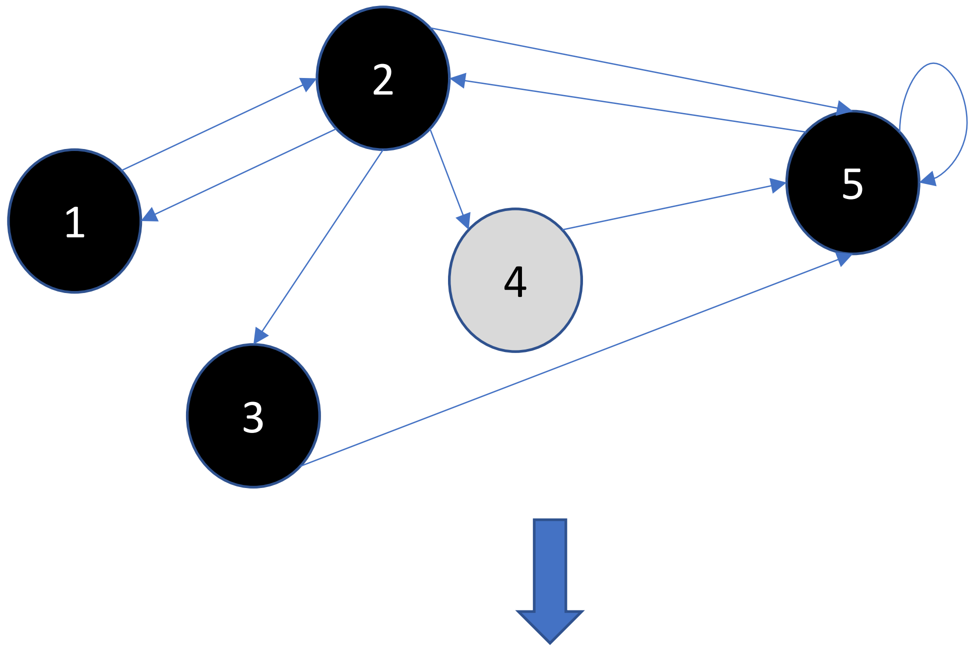
π =
{1:2,
2:5,
3:Null,
4:2,
5:3}

In our example we might want to check d[v] before appending v to Q i.e.,:

```
if d[v] < threshold: Q.append(v)
```

Breadth-first search: iteration

```
while notempty(Q):
  u = head(Q)
  for each v in Adj(u):
    if colour[v] == white:
      colour[v] = grey
      d[v]=d[u]+1
      π[v]=u
      Q.append(v)
  Q.remove(u)
  colour[u]=black
```

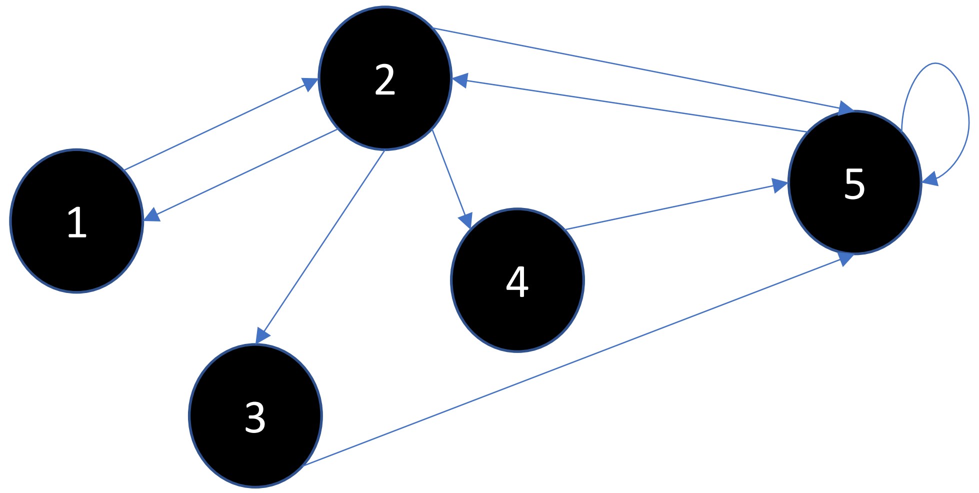


Q=[4]

Adj=	colour=	d =	π =
{1:[2],	{1:black,	{1:3,	{1:2,
2:[1,3,4,5],	2:black,	2:2,	2:5,
3:[5],	3:black,	3:0,	3:Null,
4:[5],	4:grey,	4:3,	4:2,
5:[2,5]}	5:black}	5:1}	5:3}

Breadth-first search: iteration

```
while notempty(Q):
  u = head(Q)
  for each v in Adj(u):
    if colour[v] == white:
      colour[v] = grey
      d[v]=d[u]+1
      π[v]=u
      Q.append(v)
  Q.remove(u)
  colour[u]=black
```



Q=[4]

Adj=
{1:[2],
2:[1,3,4,5],
3:[5],
4:[5],
5:[2,5]}

colour=
{1:black,
2:black,
3:black,
4:black,
5:black}

d =
{1:3,
2:2,
3:0,
4:3,
5:1}

π =
{1:2,
2:5,
3:Null,
4:2,
5:3}

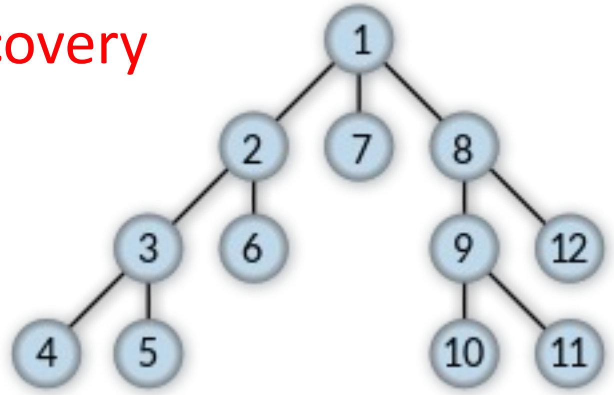
Breadth-first search: output

- d contains the shortest distance of each vertex from s
- π contains the predecessor of v on the shortest path from s to v
- It can be used to reconstruct the shortest paths / the breadth-first search tree if required
- Other functions (such as scraping the webpage and identifying links) can be called at the time that a vertex is discovered (i.e., it is coloured grey).

Breadth-first search: analysis

- Initialisation is O(V)
- The iteration happens at most V times
- In each iteration, the adjacency list for that vertex is considered
- Each edge is considered at most once throughout the whole iterative procedure
- So iteration is O(E)
- running time of BFS(G,s) = O(V+E)

Example order of discovery



Wikipedia

Depth-first search

- Alternative to breadth-first search
- Searches *deeper* in the graph whenever possible
- This can be done using a recursive call to itself

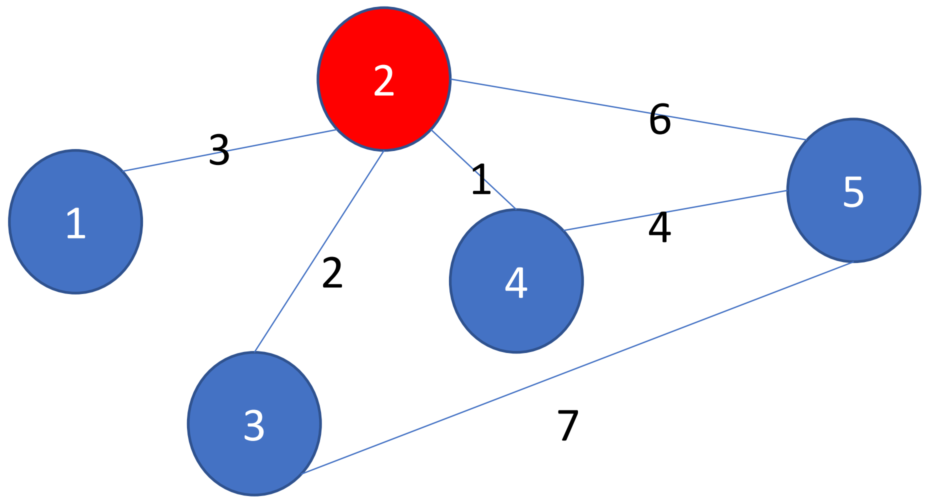
```
DFS-visit(u):
colour[u] = grey
for each v in Adj[u]:
  if colour[v] == white:
    d[v]=d[u]+1
    π[v]=u
    DFS-visit[v]
colour[u] = black
```

- Running time = O(V+E)

This recursive call means that edges from v will be explored before the remainder of the edges of u are explored.

Dijkstra's algorithm

- Solves the shortest paths problem in a weighted graph
- Modification of BFS
- Visit vertices in order of increasing distance from the source.
- As each vertex *u* is visited, check to see if the distance of each vertex *v* which is adjacent to *u* would be shorter if the path went via *u*
- We will explore this in the lab.



Overview

- Clustering.
- Current topic:
  - **Graphs / networks**

    - Breadth-first and depth-first searching DONE

    - Dijkstra's algorithm for all pairs shortest paths INTRODUCED- EXAMPLE NEXT WEEK

    - Mining social media networks NEXT WEEK
- Also next week: PageRank algorithm