

Adventures in Computer Science

COMP 102A, Lecture 22

COMP 102: Excursions in Computer Science

Sorting Data



Instructor: Joelle Pineau (jpineau@cs.mcgill.ca)

Class web page: www.cs.mcgill.ca/~jpineau/comp102

Acknowledgement: Some of today's slides were taken from:

<http://www.cs.rutgers.edu/~mlittman/courses/cs442-06/>

Sock Matching

- We've got a basketful of mixed up pairs of socks.
- We want to pair them up reaching into the basket as few times as we can.



Sock Sorter A

- Strategy: Repeat until basket is empty
 - Grab a sock.
 - Grab another.
 - If they don't match, toss them back in the basket.
 - Will this procedure ever work?
 - Will it *always* work?
-

Measuring Performance

- Let's say we have 8 pairs of socks.
- How many times does this strategy reach into the basket?
 - Min?
 - Max?
 - Average?
- How do these values change with increasing numbers of pairs of socks?

Sock Sorter B

- Strategy: Repeat until basket is empty
 - Grab a sock.
 - Is its match already on the bed?
 - If yes, make a pair.
 - If no, put it on the bed.

Measuring Performance

- Once again, assume we have 8 pairs of socks.
 - How many times does this strategy reach into the basket?
 - Min?
 - Max?
 - Average?
 - How do these values grow with increasing numbers of pairs of socks?
 - How does this compare with Sock Sorter A?
-

Comparing Algorithms

Repeat For Each Sock

sockA



sockB

- Do you have a matching pair? Set it aside.
- Do you have a non-matching pair? Put them back in the basket.
- Is there a match on the table? Pair them and set the pair aside.
- Otherwise, find an empty place on the table and set the sock down.

Notable if No Table

- Sock Sorter B seems like it is faster.
 - One disadvantage of Sock Sorter B is that you must have a big empty space.
 - What if you can only hold 2 socks at a time?
-

Sock Sorter C

- Strategy: Repeat until basket empty
 - Grab a sock.
 - Grab another.
 - Repeat until they match:
 - Toss second sock into the basket.
 - Grab a replacement.

Measuring Performance

- Once again, let's imagine we have 8 pairs of socks.
 - How many times does this strategy reach into the basket?
 - Min?
 - Max?
 - Average?
 - How do these values grow with increasing numbers of pairs of socks?
-

Comparing Algorithms

Round #2

sockA



sockC

- Do you have a matching pair? Set it aside.
 - Do you have a non-matching pair? Put them both back in the basket.
- Do you have a matching pair? Set it aside.
 - Do you have a non-matching pair? Put **one** back in the basket.

Analysis of Sock Sorter C

- Roughly the same number of matching operations as Sock Sorter A, but since it always holds one sock, roughly half the number of socks taken out of the basket.

Algorithms

- **Sock Sorter A**, **Sock Sorter B** and **Sock Sorter C** are three different algorithms for solving the problem of sock sorting.
 - Different algorithms can be better or worse in different ways.
 - **Number of operations**
E.g. total # of times reaching into basket, total # of comparisons.
 - **Amount of memory**
E.g. # of socks on the bed (or in the hand) at any given time.
-

Lessons Learned

- Given notion of **time** (# instructions to execute) and **space** (amount of memory), we can compare different algorithms.
 - **It's important to use a good algorithm!**
 - It's especially important to think **how time and space change**, as a function of the **size of the problem** (i.e. # pairs of socks).
-

Sorting Lists

- Many problems of this type! This is an important topic in CS.
 - Sorting words in alphabetical order.
 - Ranking objects according to some numerical value (price, size, ...)

Unsorted / Sorted

262, 201, 918, 301, 187, 762, 397, 277, 645, 306,
765, 798, 689, 867, 276, 402, 124, 545, 907, 569,
259, 152, 399, 481, 977, 947, 774, 727, 292, 285,
173, 599, 464, 212, 147, 696, 242, 559, 155, 569,
806, 784, 415, 321, 820, 126, 469, 225, 646, 438

124, 126, 147, 152, 155, 173, 187, 201, 212, 225,
242, 259, 262, 276, 277, 285, 292, 301, 306, 321,
397, 399, 402, 415, 438, 464, 469, 481, 545, 559,
569, 569, 599, 645, 646, 689, 696, 727, 762, 765,
774, 784, 798, 806, 820, 867, 907, 918, 947, 977

Sorting web pages

[Advanced Search](#)
[Preferences](#)

Web

[Sorting algorithm - Wikipedia, the free encyclopedia](#)

In computer science and mathematics, a **sorting algorithm** is an **algorithm** that puts elements of a list in a certain order. The most-used orders are numerical ...

en.wikipedia.org/wiki/Sorting_algorithm - 90k - [Cached](#) - [Similar pages](#)

[Quicksort - Wikipedia, the free encyclopedia](#)

Quicksort is a well-known **sorting algorithm** developed by C. A. R. Hoare that One advantage of parallel quicksort over other parallel **sort algorithms** is ...

en.wikipedia.org/wiki/Quicksort - 74k - [Cached](#) - [Similar pages](#)

[Sorting Algorithms Demo](#)

The following applets chart the progress of several common **sorting algorithms** while **sorting** an array of data using in-place **algorithms**. ...

www.cs.ubc.ca/~harrison/Java/sorting-demo.html - 11k - [Cached](#) - [Similar pages](#)

[Sorting Algorithms](#)

Description, source code, **algorithm** analysis, and empirical results for bubble, heap, insertion, merge, quick, selection, and shell sorts.

linux.wku.edu/~lamonml/algor/sort/sort.html - 9k - [Cached](#) - [Similar pages](#)

[Sorting Algorithms](#)

Shows the number of comparisons, performed by the **sorting algorithm**. ... 4. Shows the code listing of the performed **sorting algorithm**. ...

maven.smith.edu/~thiebaut/java/sort/demo.html - 3k - [Cached](#) - [Similar pages](#)

[Sorting Algorithms](#)

Overview of many **sorting** techniques and corresponding links.

www.softpanorama.org/Algorithms/sorting.shtml - 67k - [Cached](#) - [Similar pages](#)

Sorting arrays

- Consider an array containing a list of names:

Lindsey

Christopher

Nicholas

Erica

Rahul

Jane

- How can we arrange them in alphabetical order?
-

A simple way to sort: Bubble sort

- Compare the first two values. If the second is larger, then swap. **smaller**
- Continue with the 2nd and 3rd values, and so on.
- When you get to the end, start again.
- Repeat until no values are swapped.

Original list:

Lindsey
Christopher
Nicholas
Erica
Rahul
Jane

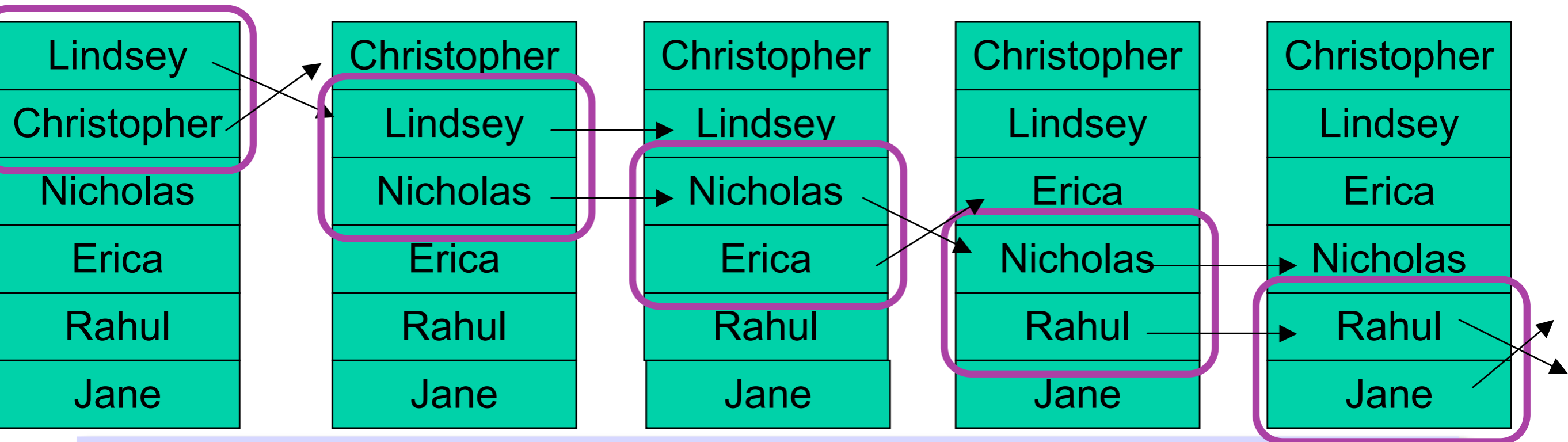
Partially sorted list:

Christopher
Lindsey
Nicholas
Erica
Rahul
Jane

Christopher
Lindsey
Nicholas
Erica
Rahul
Jane

Christopher
Lindsey
Erica
Nicholas
Rahul
Jane

Christopher
Lindsey
Erica
Nicholas
Rahul
Jane



Let's think about Bubble sort

- Is this a good way to sort items?
 - Simple to implement. This is good!
 - Guaranteed to find a fully sorted list. This is good too!
- How do we decide whether it's a good method?

Useful things to consider

- How long will it take?
 - How much memory will it take?
 - Is there a way we can measure this?
 - Best criteria:
 - number of basic machine operations: *move, read, write data*
 - amount of machine memory
 - Can we think of something similar, at a higher level?
-

Predicting the “cost” of a sorting program

- Number of pairwise comparisons

For Bubble sort:

- Let's say n is the **number of items** in the array.
- Need $n-1$ **comparisons** on every pass through the array.
- Need n **passes** in total (at most).
- So $n*(n-1)$ **pairwise comparisons**.

- Amount of memory we need (in addition to the original array)

For Bubble sort:

- Everything happens within the original array.
 - Need to keep track of the **index of the current item** being compared.
 - Need to keep track, during each pass, of **whether a swap was done**.
 - So **only 1 integer and 1 bit of memory**.
-

A more intuitive sort method: Selection sort

- Scan the full array to find the first element, and put it into 1st position.
- Repeat for the 2nd position, the 3rd, and so on until array is sorted.

Original list:

Lindsey
Christopher
Nicholas
Erica
Rahul
Jane

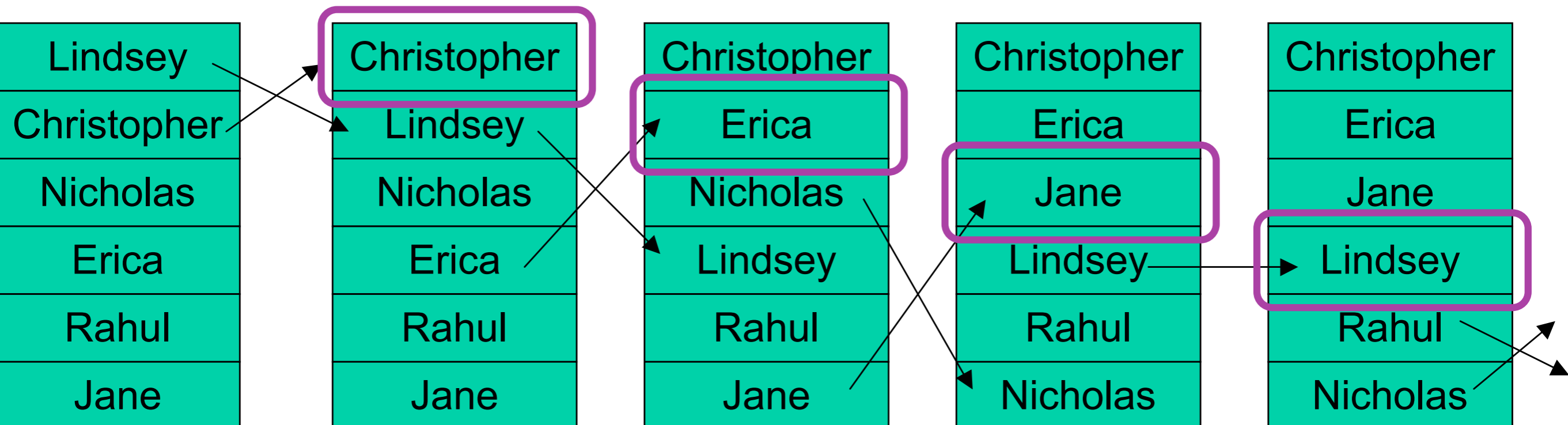
Partially sorted list:

Christopher
Lindsey
Nicholas
Erica
Rahul
Jane

Christopher
Erica
Nicholas
Lindsey
Rahul
Jane

Christopher
Erica
Jane
Lindsey
Rahul
Nicholas

Christopher
Erica
Jane
Lindsey
Rahul
Nicholas



What is the “cost” of Selection sort?

- Number of pairwise comparisons
 - Let's say n is the number of items in the array.
 - Need $n-1$ comparisons on the 1st pass through the array.
 - Need $n-2$ comparisons on the 2nd pass through the array.
 - And so on until we reach the last two elements.
 - So in total: $(n-1) + (n-2) + (n-3) + \dots + 1 = n * (n-1) / 2$ pairwise comparisons.
 - This is better than Bubble sort. (But only by a factor of 2.)
 - Amount of memory we need (in addition to the original array)
 - Everything happens within the original array.
 - Need to keep track of the index of the current item being compared.
 - Need to keep track, during each pass, of the index of the best value found so far.
 - So only 2 integers in memory. Roughly the same as Bubble sort.
-

Why do we care about the “cost”?

- Need to know whether we can use our program or not!
 - Can we use Selection sort to alphabetically sort the words in the English Oxford dictionary?
 - About 615,000 entries in the 2nd edition (1989).
 - So we would need 189 trillion pairwise comparisons!
 - What if we try to sort websites according to hostnames:
 - About 127.4 million active domain names (as of January 2011).
 - So we would need $8.06 \cdot 10^{15}$ pairwise comparisons!
 - Fortunately, not much “extra” memory is needed :-))
-

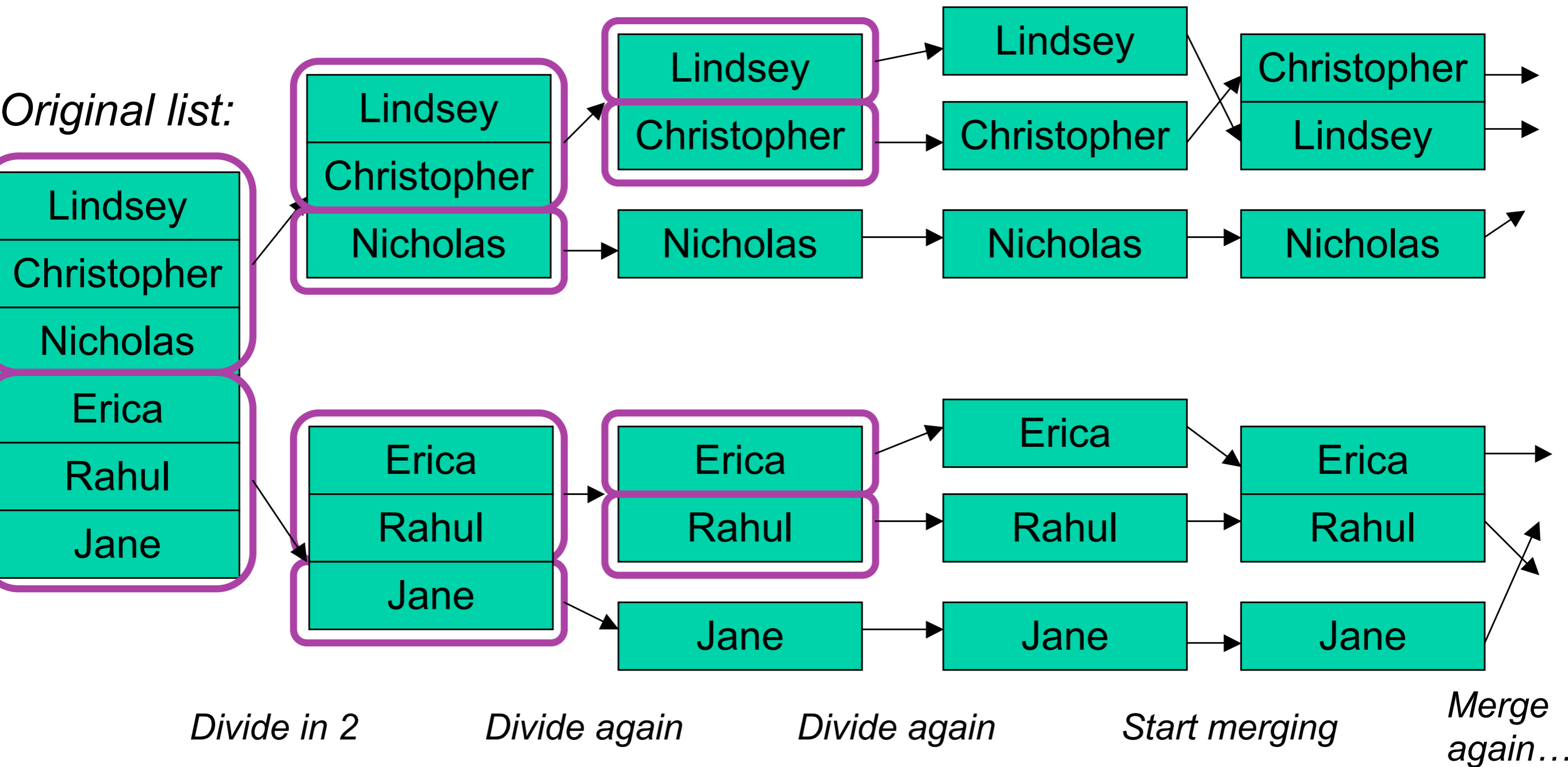
Let's find a better way: Merge sort

- Divide-and-Conquer! (This is our old friend “Recursion”.)
 - Main idea:
 1. Divide the problem into subproblems.
 2. Conquer the sub-problems by solving them recursively.
 3. Merge the solution of each subproblem into the solution of the original problem.
 - What does this have to do with sorting?
-

Merge sort

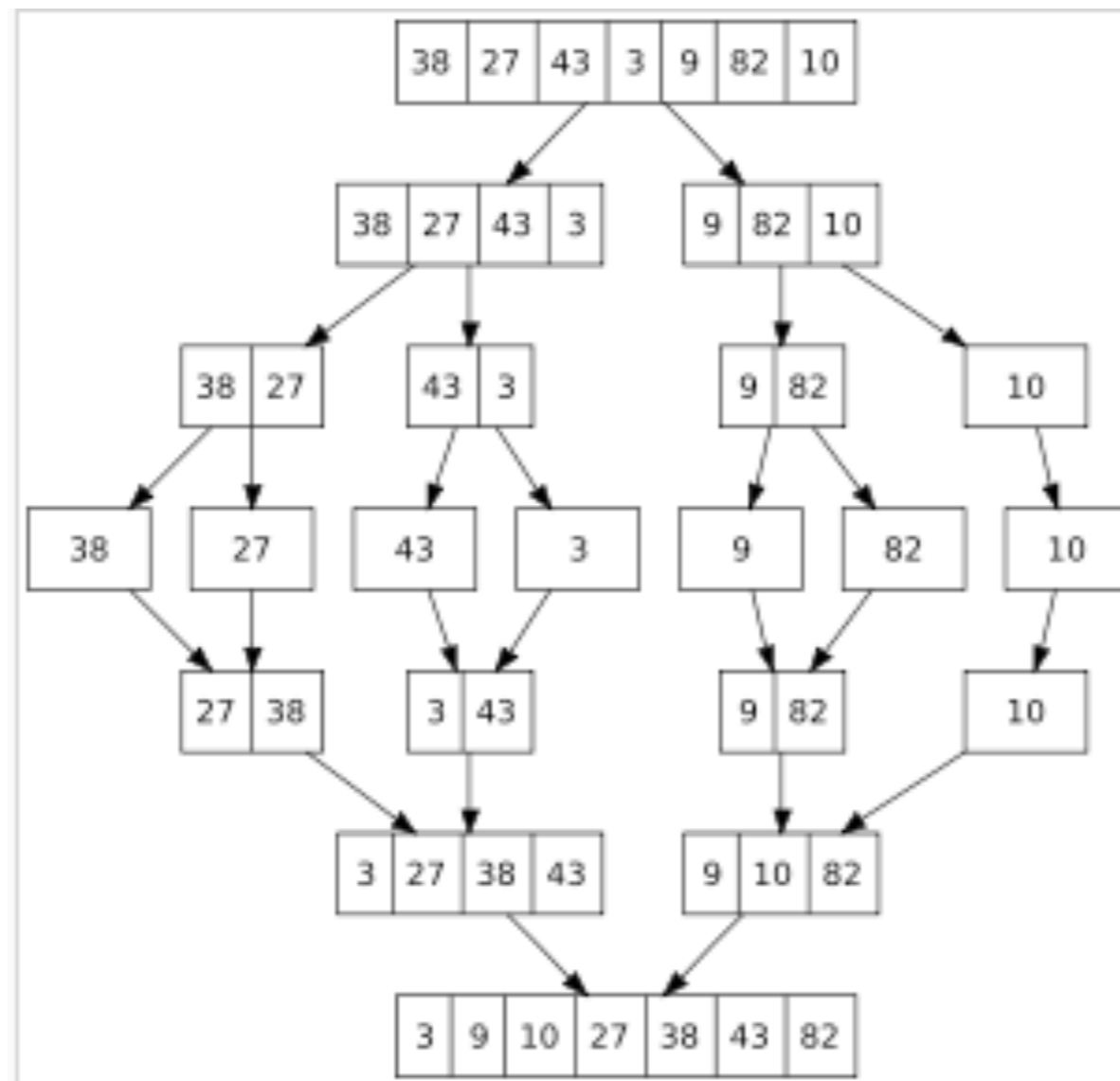
- Example:
 - Sort an array of names to be in alphabetical order.
 - Algorithm:
 1. Divide the array into left and right halves.
 2. Conquer each half by sorting them (recursively).
 3. Merge the sorted left and right halves into a fully sorted array.
-

Merge sort: An example



Another example of Merge sort

- Consider sorting an array of numbers:



Let's think about Merge sort

- Possibly harder to implement than Bubble sort or Selection sort.
 - Number of pairwise comparisons:
 - How many times we **divide into left/right sets**? At most $\log_2(n)$
 - How many **items to sort** once everything is fully split? **None!**
 - How many **comparisons during merge**, if subsets are sorted?
 - Need about n **comparisons** if sorted subsets have $n/2$ items each.
 - So in total: n comparisons per level * $\log_2(n)$ levels = $n * \log_2(n)$
 - **This is better than Bubble sort and Selection sort (by a lot).**
 - Amount of memory we need (in addition to the original array):
 - Every time we merge 2 lists, we need extra memory.
 - For the last merge, we need a full n -**item array of extra memory**.
 - This is **worse than Bubble sort and Selection sort**, but not a big deal.
 - We also need **2 integers** (1 for each list) to keep track of where we are during merging.
-

Merge sort is a bargain!

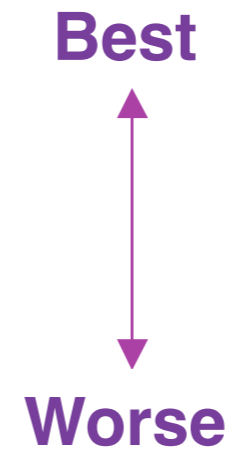
- Using Merge sort to alphabetically sort the words in the English Oxford dictionary.
 - Recall: about 615,000 entries in the 2nd edition (1989).
 - So we would need 11.8 million pairwise comparisons.
 - Versus 1.89 trillion if using Selection sort!
- Using Merge sort to organize websites according to hostnames:
 - Recall: about 127.4 million active domain names (as of January 2011).
 - So we would need 3.4 billion pairwise comparisons.
 - Versus $8.06 \cdot 10^{15}$ if using Selection sort!

Number of comparisons

- Between Dec. 2007 and Jan. 2011 number of domains names grew from 62 millions to 127 millions.
 - Number of comparisons with Bubblesort grows from $3.4 \cdot 10^{15}$ to $1.6 \cdot 10^{16}$.
 - Number of comparisons with Mergesort grows from 1.6 to to 3.4 billion comparisons.

Quick recap on the number of operations

- Number of operations (y) as a function of the problem size (n)
 - Constant: $y = c$
 - Linear: $y = n$
 - Log-linear: $y = n \cdot \log_2(n)$
 - Quadratic: $y = n^2$
 - Exponential: $y = 2^n$
- Bubble sort and Selection sort take a **quadratic** number of comparisons.
 - This is as bad as it gets, for sorting algorithms. (**...for natural algorithms...**)
- Merge sort takes a **linear*log** number of comparisons.
 - This is as good as it gets, for sorting algorithms.
- This is a worst-case analysis (i.e. maximum number of operations.)



A word about memory

- Merge sort uses twice as much memory as Selection sort.
 - This is not a big deal. If you can store the array once, you can probably store it twice.
- But computers have 2 types of memory:
 - RAM (rapid-access memory) and hard-disk memory.
 - RAM is much faster, but usually there is less of it.
 - As long as everything fits into RAM, no problem!
- If array is too large for RAM, then you need to worry about:
 - Number of times sections of the array are copied / swapped to and from disk.

Take-home message

- Sorting is one of the **most useful algorithms**.
 - Applications are everywhere.
 - There are **many ways to solve a problem**.
 - For sorting: **Bubble sort, Selection sort, Merge sort**, and many more.
 - Some methods use $n \cdot \log_2(n)$ comparisons and (almost) no extra memory!
 - When choosing an algorithm to solve a problem, it's important to think about the **cost (= time and memory)** of this algorithm.
 - It's also useful to think about how "easy" the algorithm is to program (more complicated = more possible mistakes), but this is harder to quantify.
-





COMP 102: Excursions in Computer Science

Searching



Instructor: Joelle Pineau (jpineau@cs.mcgill.ca)

Class web page: www.cs.mcgill.ca/~jpineau/comp102

Searching example

- **Given:** A list of names of students and their favourite colour.
- **Problem:** Find the favourite colour of the student named Alice, if she is in the class.

<ol style="list-style-type: none">1. Bob 'black'2. Mary 'red'3. Carol 'yellow'4. Allison 'blue'5. Alice 'yellow'6. Joe 'green'7. Joseph 'purple'

Sequential Search

- Process each list entry from first to last.
 - Check if each entry processed is the entry for “Alice”.
 - If we find the “Alice” entry,
 - Note Alice’s favourite colour.
 - Stop searching.

- How many entries in the list are processed before Alice is found?

1. Bob ‘black’
2. Mary ‘red’
3. Carol ‘yellow’
4. Allison ‘blue’
5. Alice ‘yellow’
6. Joe ‘green’
7. Joseph ‘purple’

Sequential search

- How many entries in the list are processed before Alice is found?

1. Bob 'black'
2. Mary 'red'
3. Carol 'yellow'
4. Allison 'blue'
5. George 'green'
6. Billy 'white'
7. Walter 'yellow'
8. Geoffrey 'pink'
9. Alice 'yellow'
10. Joe 'green'
11. Joseph 'purple'

Sequential Search on a Sorted List

- Can you speed-up the search if the list is sorted?

Yes! If you are looking for Alice.

What if you are looking for Joe's favourite colour?

Sorting won't help. Or can it?

1. Alice 'yellow'
2. Allison 'blue'
3. Bob 'black'
4. Carol 'yellow'
5. Joe 'green'
6. Joseph 'purple'
7. Mary 'red'

Binary Search

- Search algorithm for sorted lists.
- How do you find a word in the dictionary?

E.g. “Joe”

Binary Search on a Dictionary

- Look at the **middle** page of the dictionary.
 - Read the words on this page.
 - If the word you are looking for comes **after** these words:
 - Search among the pages of the dictionary that come after this page.
 - If the word you are looking for comes **before** these words:
 - Search among the pages of the dictionary that come before.
 - If the word you are looking for is **on** this page,
 - Stop searching!
-

Binary Search for “Joe”

1. Alice ‘yellow’

2. Allison ‘blue’


3. Bob ‘black’

First, try the middle. 

4. Carol ‘yellow’

Third try, got it! 

5. Joe ‘green’

Second, try the middle
of the second half. 

6. Joseph ‘purple’




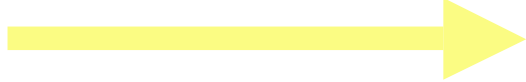
7. Mary ‘red’

Comparing Search Algorithms

- Sequential search: 5 items examined to find “Joe”.
- Binary search: 3 items examined to find “Joe”.

- Which would choose?

Binary Search for “Walter”

1. Alice ‘yellow’
 2. Allison ‘blue’
 3. Billy ‘white’
 4. Bob ‘black’
 5. Carol ‘yellow’
 - 1st try  6. Geoffrey ‘pink’
 7. George ‘green’
 8. Joe ‘green’
 - 2nd try  9. Joseph ‘purple’
 - 3rd try  10. Mary ‘red’
 - 4th try  11. Walter ‘yellow’
-

Comparing Search Algorithms

- Searching for “Joe”:
 - Sequential search: 5 items examined.
 - Binary search: 3 items examined.
 - Searching for “Walter”
 - Sequential search: 11 items examined.
 - Binary search: 4 items examined.
 - Which would choose?
-

Worst-Case Analysis

- Binary search seems faster than sequential search for sorted lists.
 - Let's think about the maximum possible number of items we need to check.
 - With **sequential search**: **N elements**

where **N** = numbers of items in the sorted list.

“ If there are 7 elements in the list, then in the worst-case, sequential search looks at 7 elements before finding the answer. “
 - With **binary search**: ???
-

Worst-case Complexity of Binary Search

- Here, at most **3** elements of the list need to be analyzed.

1. Alice 'yellow'

2. Allison 'blue'

3. Bob 'black'



4. Carol 'yellow'



5. Joe 'green'




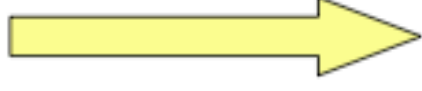


6. Joseph 'purple'

7. Mary 'red'

Worst-case Complexity of Binary Search

- Here, at most **4** elements of the list need to be analyzed.

1. Alice 'yellow'
2. Allison 'blue'
3. Billy 'white'
4. Bob 'black'
5. Carol 'yellow'
-  6. Geoffrey 'pink'
7. George 'green'
8. Joe 'green'
-  9. Joseph 'purple'
-  10. Mary 'red'
-  11. Walter 'yellow'

Why should you care?

- If your database has 8,388,607 names (e.g. the telephone book), using **sequential search** may examine all 8,388,607 names.
- To search a list of 8,388,607 names using **binary search** examines how many names at most?

How do we get this?

If you have 1 names in the list, need at most 1 check.

If you have 2 names in the list, need at most 2 checks.

If you have 4 names in the list, need at most 3 checks.

If you have 8 names in the list, need at most 4 checks.

If you have 16 names in the list, need at most 5 checks.

.....

If you have N names in the list, need at most $\log_2(N)+1$ checks.

But!

- Binary search only works on **sorted lists**.
 - In the worst-case, if you sort using Bubble sort, you will need:
 $n*(n-1)$ comparisons for **Bubble sort** + $\log_2(n)$ comparisons for **Binary search**
 - In the worst-case, if you sort using Merge sort, you will need:
 $n*\log_2(n)$ comparisons for **Merge sort** + $\log_2(n)$ comparisons for **Binary search**
 - So why not keep things simple and use:
 n comparisons for **Sequential search** (no sorting necessary) ?
-

Binary search vs Sequential search

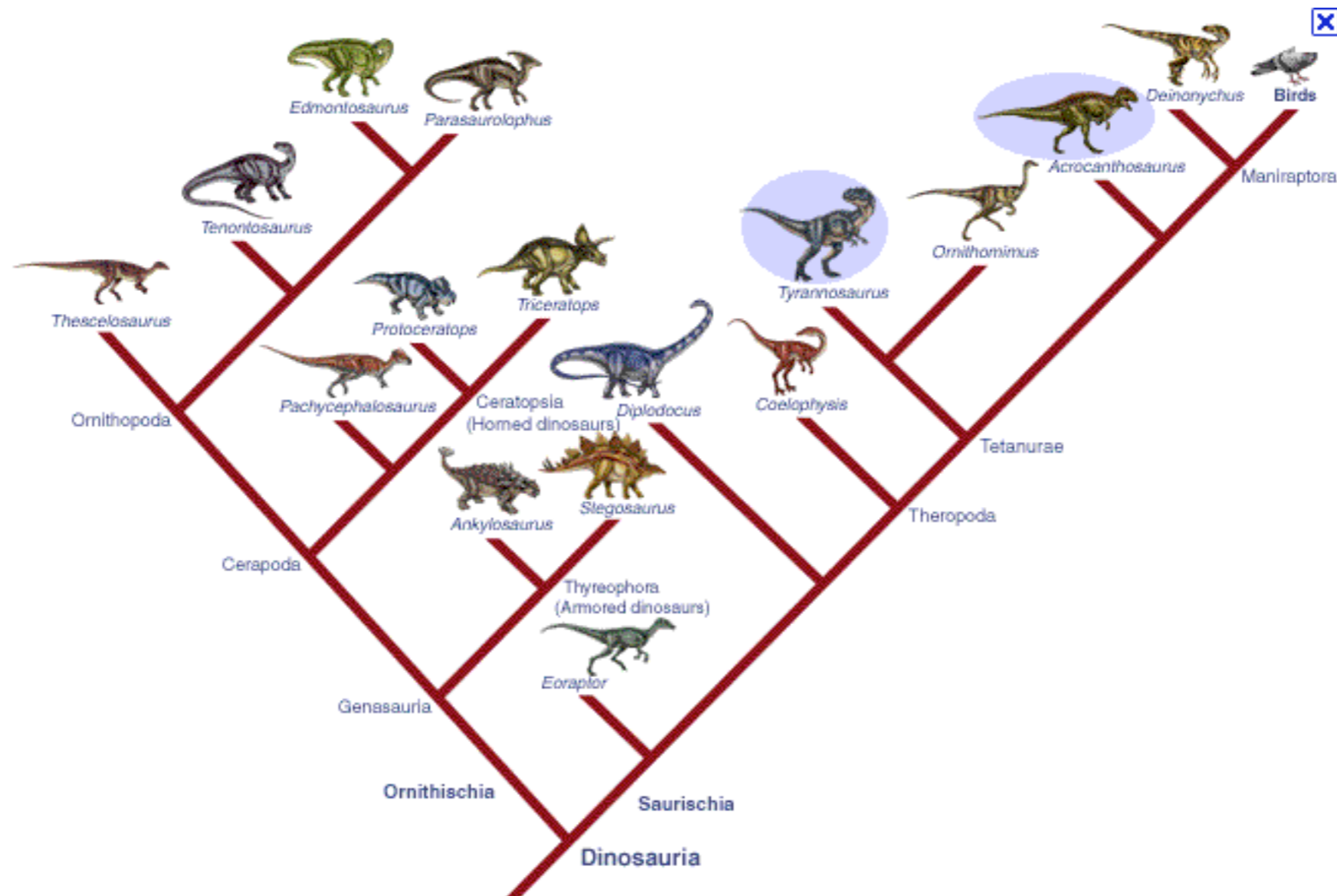
- In general, you need to **sort only once**, and then you can **search** the sorted list as **many times** as we want.
- If you don't need to do **multiple searches**, then it is better to just do sequential search, without any pre-sorting.

Quick Recap

- Searching is as useful as (if not more than) sorting.
 - So far we have seen searching on arrays (sorted or not).
 - This is interesting, but the fun is only beginning!
 - In many problems, data is not stored in an array.
-

Searching data organized in a tree

- You excavated a fossil, and are trying to identify its species.
 - In what **order** do you consider the nodes in the tree?



Searching through a maze

- Interesting questions:
 - How do we search through the maze?
 - How do we encode this problem for the computer?



Searching through the subway system

What is the **shortest path** from the Université de Montréal station to the McGill station?

How should we encode this problem?

Can't store the list of stations in a simple array.

This is an example of a **graph**.



Graphs

A **graph** is an abstract representation defined by a pair **(N, E)**, where

N is a collection of **nodes** (or objects)

E is a collection of pairs of nodes, called **edges** (representing the relations between the objects.)

In the Montreal metro system:

- What are the nodes?

The metro stations.

- What are the edges?

Rail link between neighbouring stations.

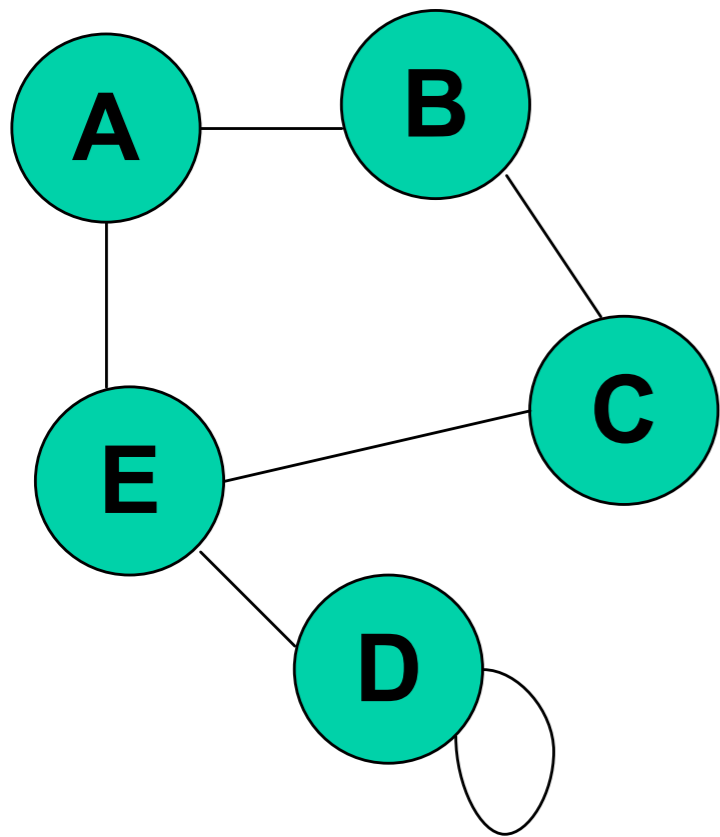
Paths

- A **path** is a sequence of adjacent nodes.
E.g. “McGill” - “Place-des-Arts” - “St-Laurent” - “Berri-UQAM”
 - The **path length** is the total number of nodes along a path.
 - We can store a graph in memory using an **adjacency matrix**, which defines which nodes are next to each other.
-

Adjacency matrix

- Consider a 2-D matrix, showing the relation between any pair of nodes (1=neighbours, 0=not neighbours).

Example

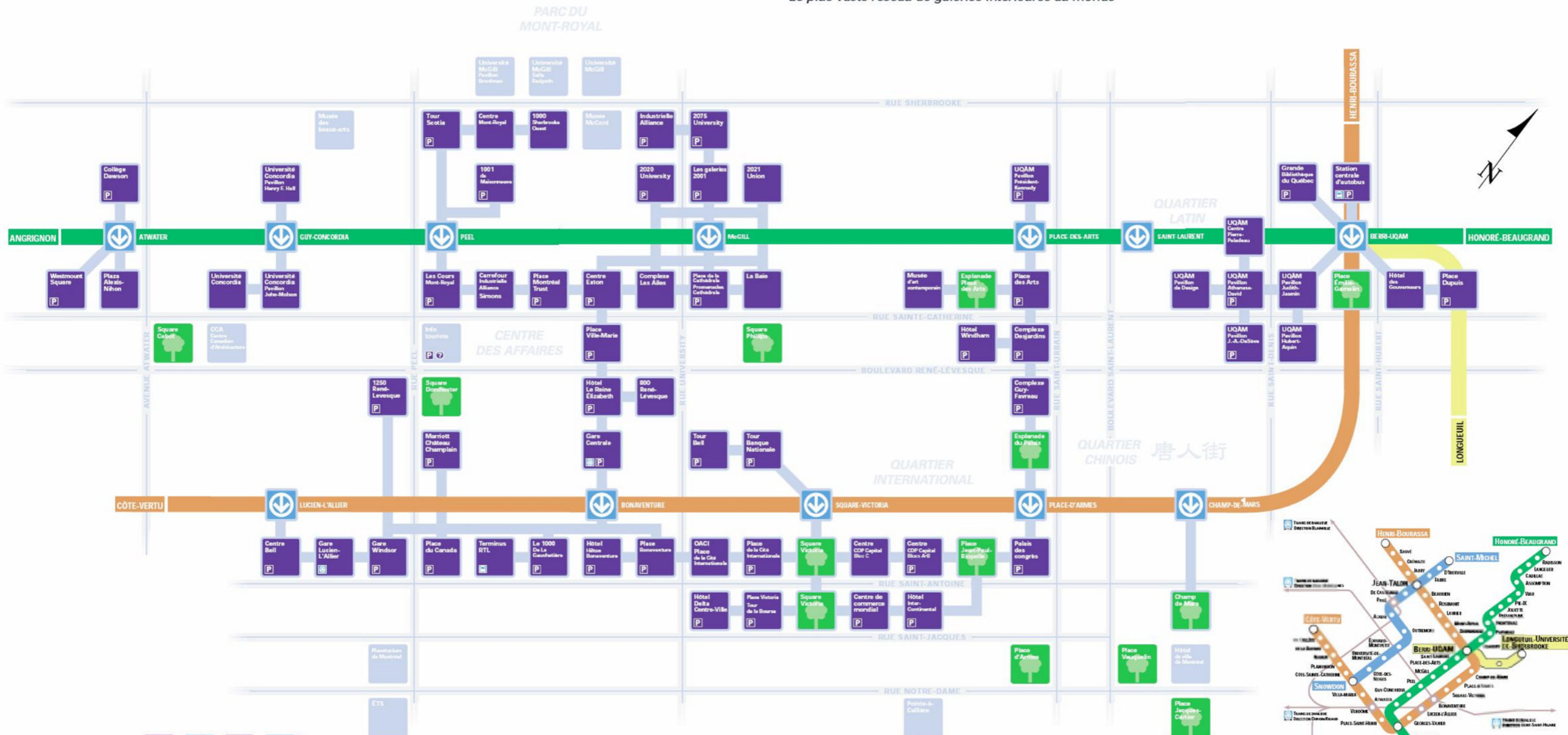


	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	0	1
D	0	0	0	1	1
E	1	0	1	1	0

Downtown Montreal map



Le plus vaste réseau de galeries intérieures au monde



CARTE-SCHEMA DU RÉSO ET DE CERTAINS LIEUX D'INTÉRÊT ET SERVICES DE MONTREAL

- Immeuble et lien RESO
- Lieu d'intérêt et services
- Parc
- Station de métro
- Trains de banlieue et gare
- Terminis d'autobus
- Stationnement
- Info touristique



Interesting questions on graphs

Question #1: What is the shortest path between two given (non-neighbour) nodes?

Question #2: What is the best path to visit all nodes with minimum overall travel time?

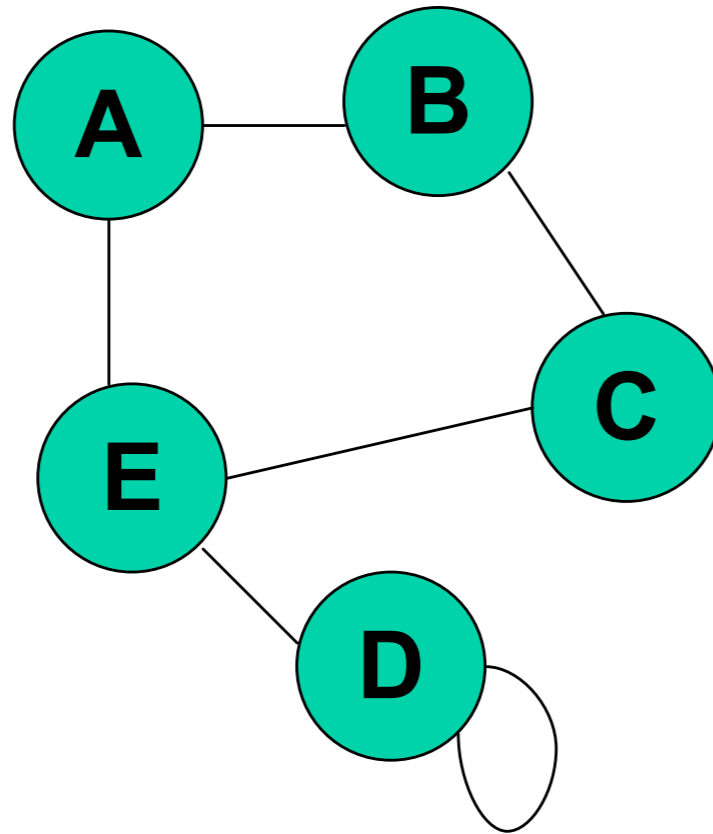
Many more interesting questions!

A few more definitions

- A **directed graph**, is a graph where there may be an edge from A to B, but not from B to A. So we say there is a **direction** to each edge.
 - In **undirected graphs**, each connected pairs of nodes is **connected in both directions**.
 - A **cycle** is a path in which the **first and last nodes are the same**.
 - A **tree** is a graph that has **no cycle**.
-

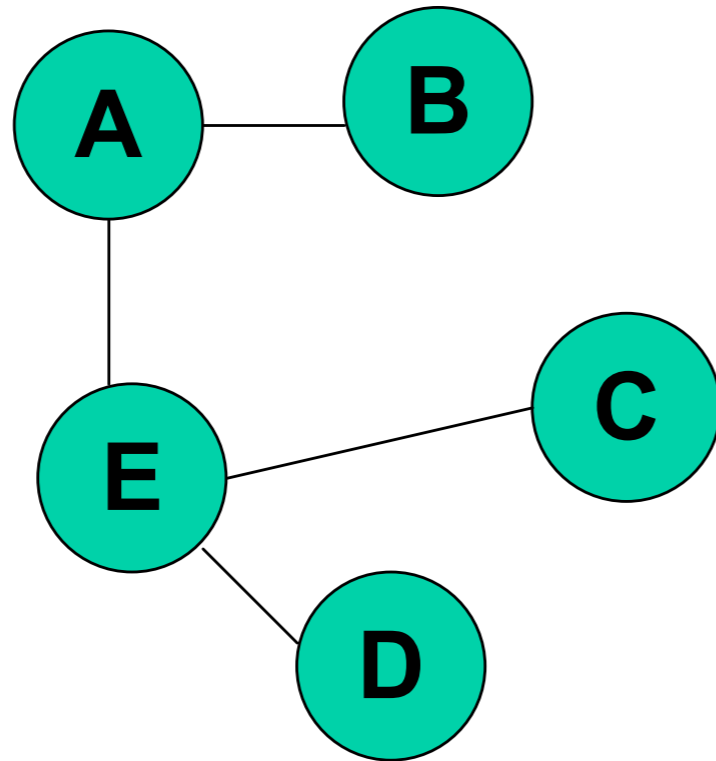
Example of a Cycle

- Nodes A-B-C-E form a cycle.
- Node D forms a cycle.



Example of a Tree

- The following graph is a tree.



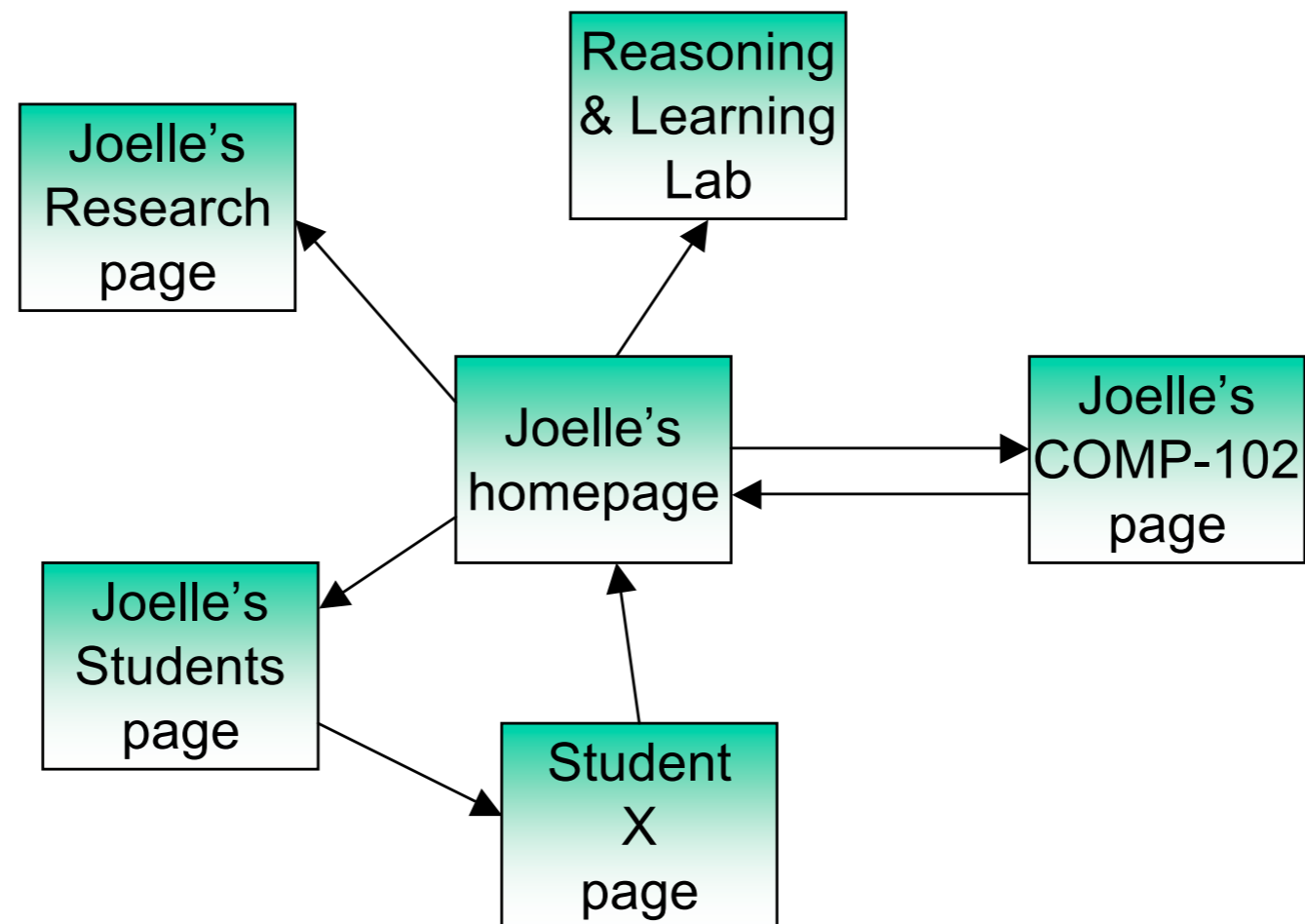
Example of an Undirected Graph

If A is a neighbour of B,
then B is a neighbour of A
(and similarly for all nodes.)



Example of a Directed Graph

- The internet!
 - Nodes are the web-pages.
 - Edges are the hyper-links, taking you from one page to another.



Take-home message

- Searching is one of the most useful algorithms.
 - You should understand sequential search and binary, and be familiar with the pros/cons of each.
 - Be able to recognize graphs, and define the key components (nodes, edges, paths, etc.)
-

COMP 102: Excursions in Computer Science

Graphs

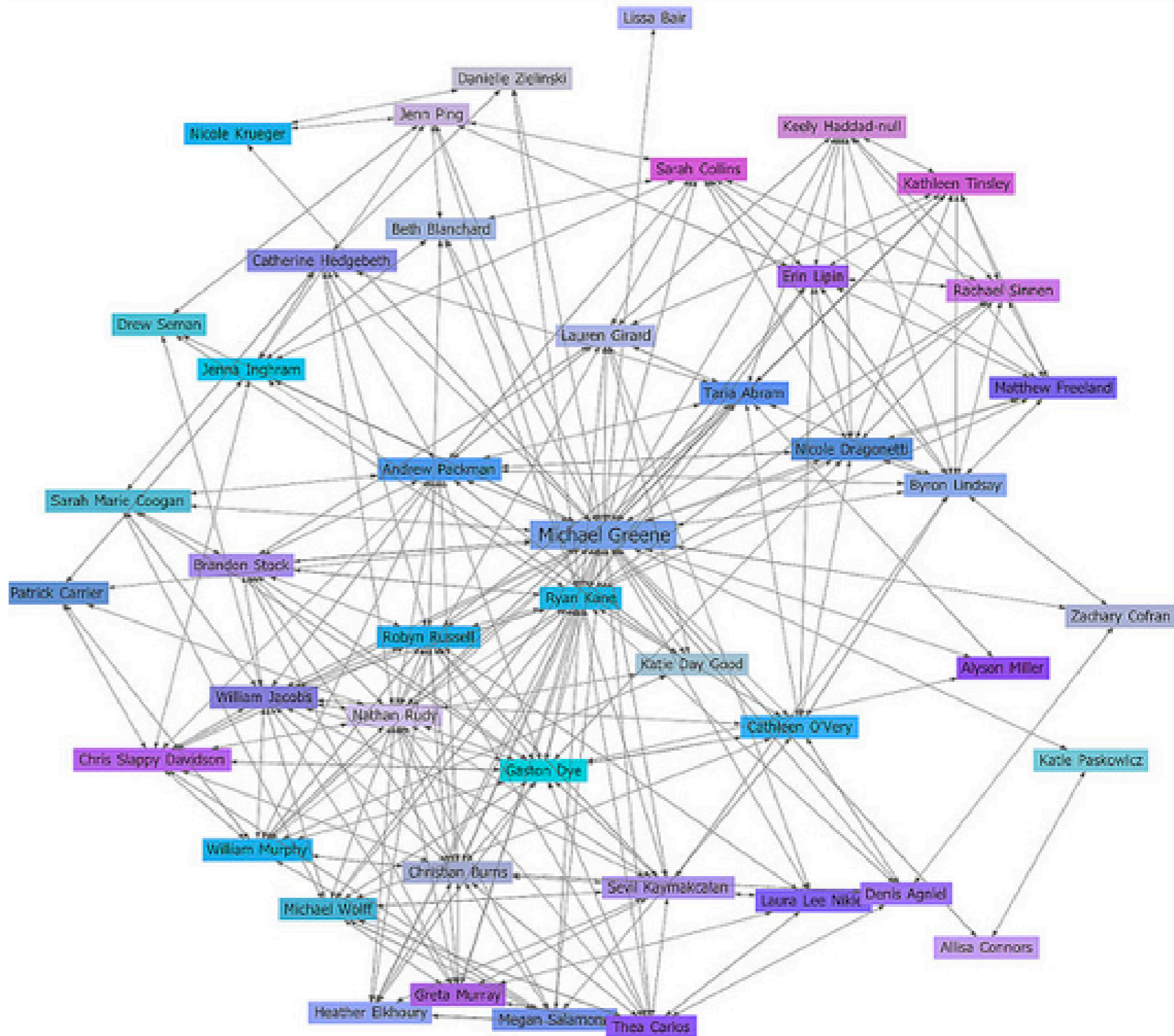


Instructor: Joelle Pineau (jpineau@cs.mcgill.ca)

Class web page: www.cs.mcgill.ca/~jpineau/comp102

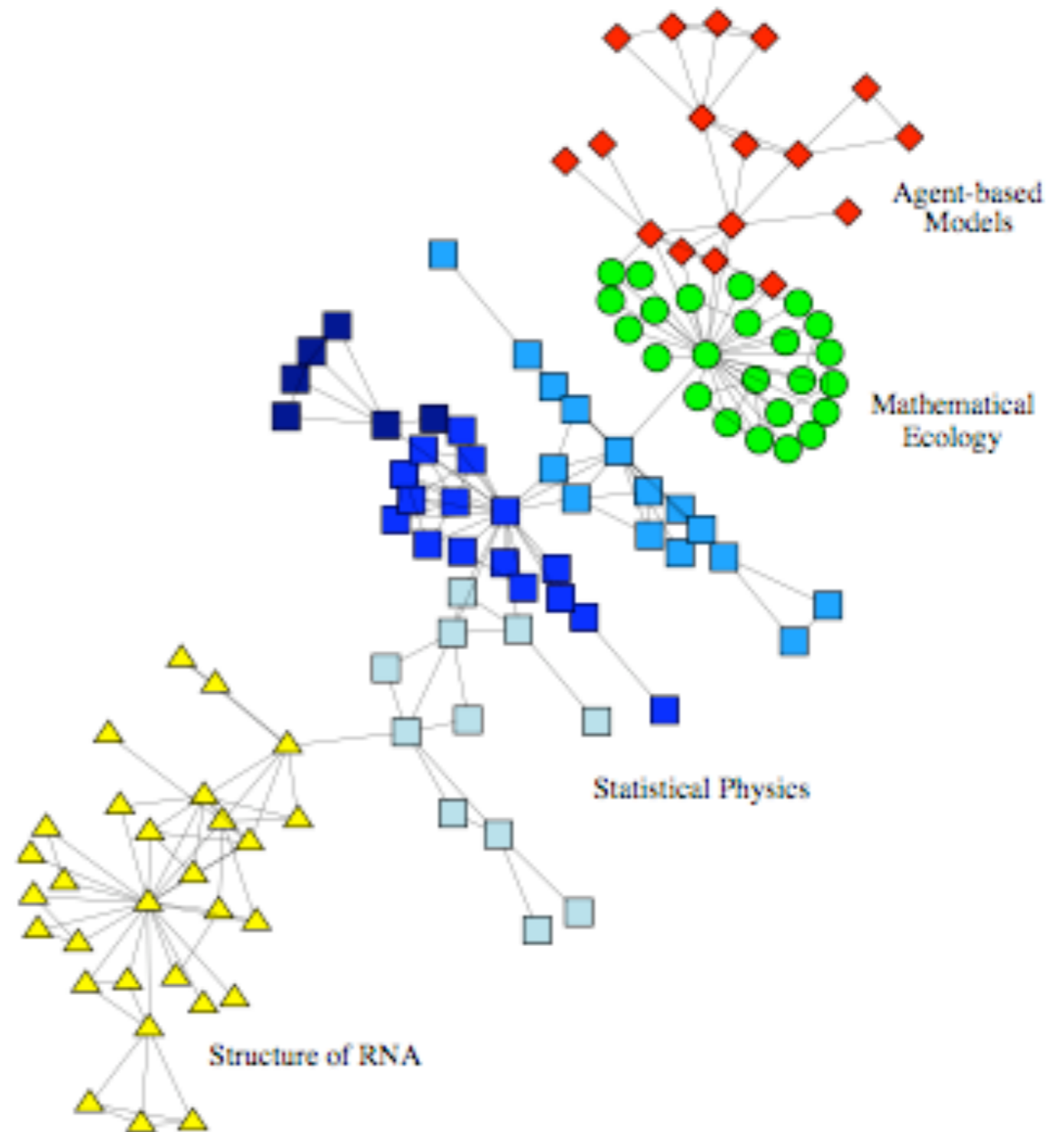
Example: A friendship network

- Graphs are sometimes also called **networks**.
- Graph analysis tool on Facebook to analyze patterns of friendships.
 - **Nodes**: people
 - **Edges**: friendships
- Could annotate the types of relationships.



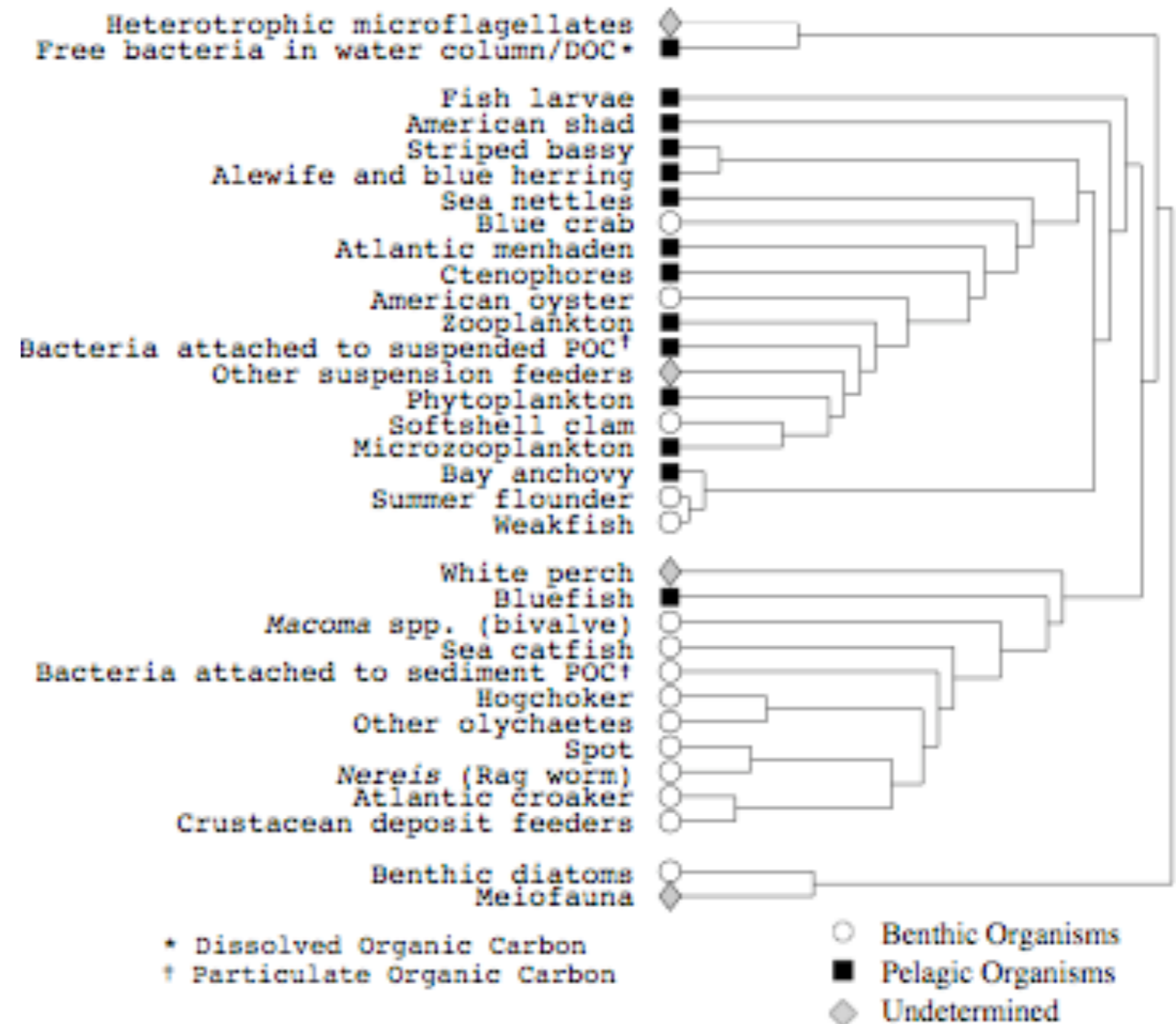
Example: Scientific collaborations

- Nodes correspond to scientists in residence at the Santa Fe Institute in 1999-2000, and their collaborators.
- An edge is drawn between a pair of scientists if they coauthored one or more articles during this time period.
- The research topics are shown as different colours. These are identified automatically using a *clustering* algorithm.



Example: Food web

- Nodes correspond to the most prevalent marine organisms living in the Chesapeake Bay (USA).
- An edge is drawn between a pair if one of the organisms eats the other.
- Graph suggests there are two well-defined communities.
- These correspond quite closely to **pelagic** organisms (those that live near the surface) and **benthic** organisms (those that live near the bottom).



Searching over Graphs

- Your graph is defined by a set of nodes and an adjacency matrix.
- You also need to know the start node and the end node.
- The goal is to explore all possible paths and return the shortest one.

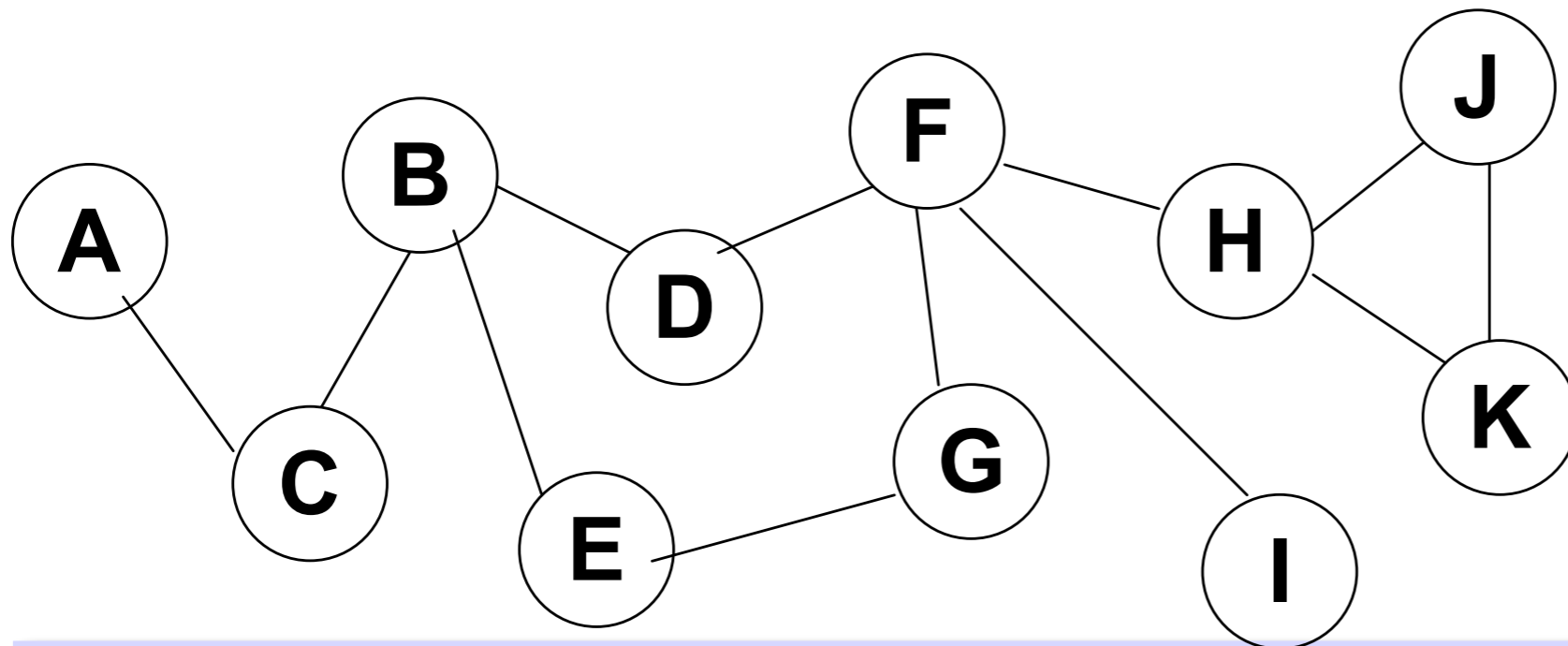
Warning! Need to be systematic about the order in which you explore these paths.

Breadth-first search

- Start at some node n. Say we start with F.
- Explore all the neighbors of n. Explore D, G, I and H.
- Then explore all the unvisited neighbours of the neighbours of n. B, E, K, J
- Then visit unvisited neighbours of those. C
- Continue until no more unvisited nodes remain. A

Visitation order: F, D, G, I, H, B, E, K, J, C, A

Visitation path: F - D - F - G - F - I - F - H - F - D - B - D - F - G
- E - G - F - H - K - H - J - H - F - D - B - C - A



Comments on breadth-first search

- Breadth-first search explores the graph **layer by layer**.
 - E.g. For web-browsing, all n-away links are explored.
- **IMPORTANT:**
 - Need to decide before-hand on the order of neighbours (e.g. clockwise)
 - Need to keep track of nodes you've already explored.
- **Pro:** Good algorithm if you want to find the **shortest path** between the start node, and another node. (As soon as you find that node, you know you have found the shortest path to it.)
- **Con:** Often requires a lot of **backtracking** (= visitation path goes through visited nodes again and again.)

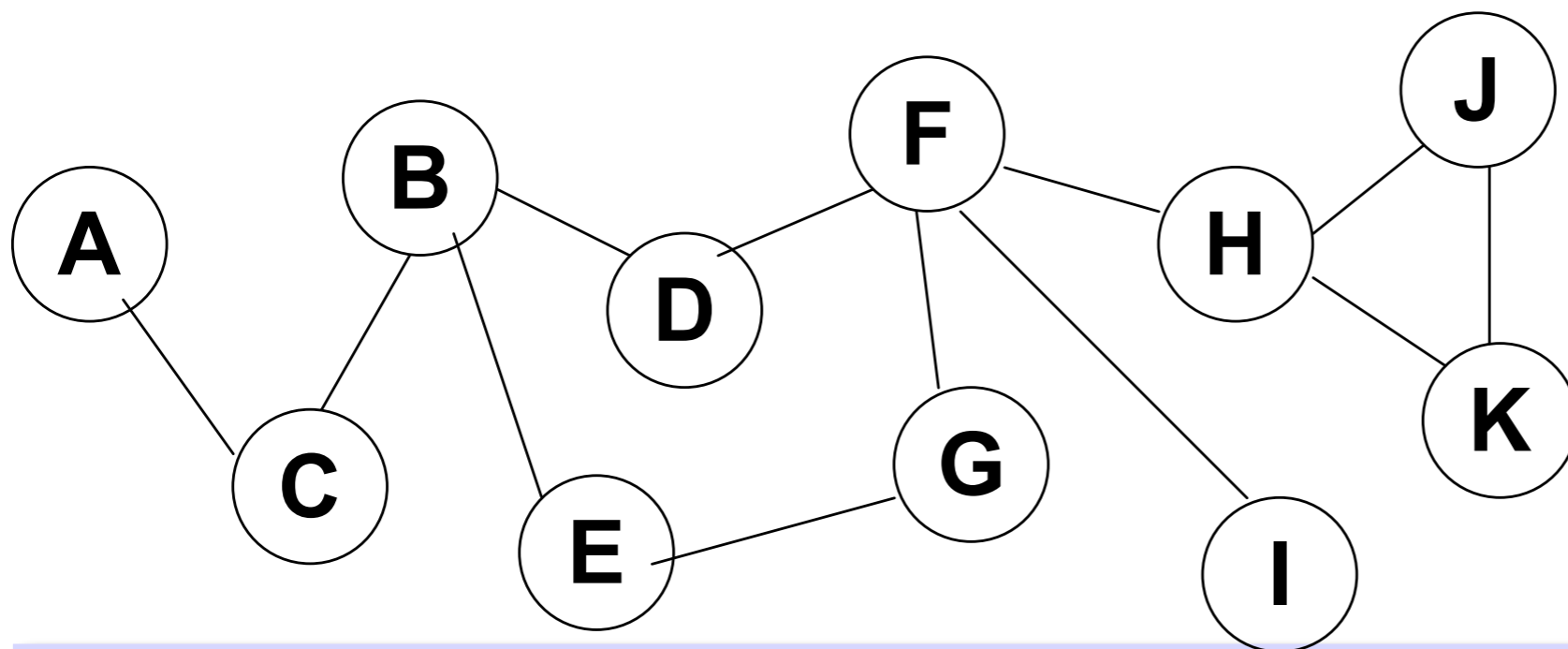
Can we avoid all this backtracking?

Depth-first search

- Start at some node n. Say we start with F.
- Then explore the first unvisited neighbour of n (call this n'). Explore D.
- Then explore the first unvisited neighbour n', and so on until you hit a node with no unexplored neighbours. B, C, A
- Then backtrack 1 level to explore the next unvisited neighbour. E, G, etc.

Visitation order: F, D, B, C, A, E, G, I, H, K, J

Visitation path: F - D - B - C - A - C - B - E - G - E - B - D - F
- I - F - H - K - J



Comments on depth-first search

- Depth-first search explores graph by going **deeper whenever possible**.
E.g. For web-browsing, always click on 1st link until you hit a dead-end.
IMPORTANT:
 - Need to decide before-hand on the order of neighbours (e.g. clockwise)
 - Need to keep track of nodes you've already explored.
 - **Pro:** Usually uses much **less backtracking** to explore the full graph than breadth-first search. How much less depends on neighbourhood ordering (sometimes lucky, sometimes not)
 - **Con:** Not guaranteed to find the shortest path, unless you explore the full graph.
 - E.g. After 3 rounds, found path to "E": F-D-B-E, which is longer than F-G-E.
-

Can we try a Best-first search?

- Start at some node n .
- Pick a score function.
- Add its neighbours to the list of candidate nodes.
- Pick candidate node with best score.
- Add its neighbours to the list of candidate nodes.
- Continue until no more unexplored nodes.

Say we start with F.

Say score = alphabetical order.

Add D(=4), G(=7), I(=9), H(=8).

Pick D.

Add B(=2).

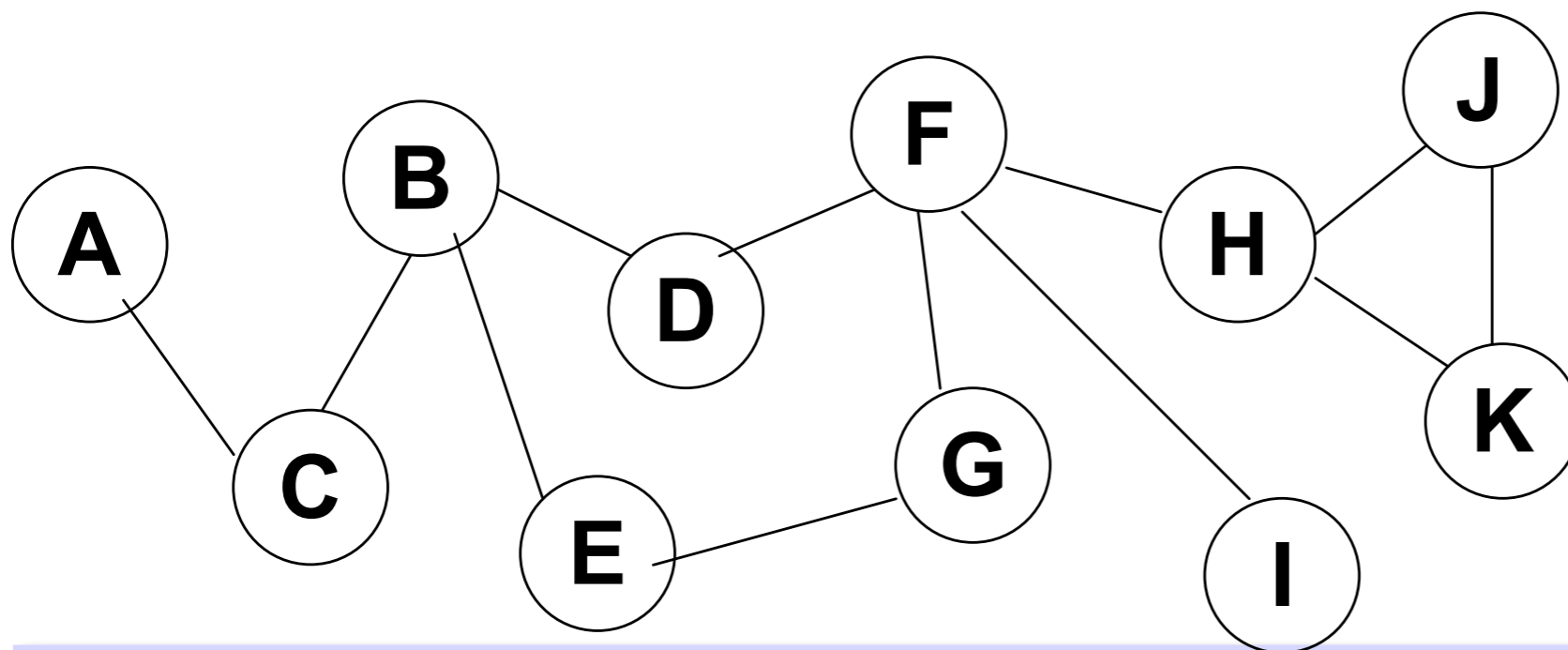
Pick B, Add C(=3) and E(=5), etc.

Exploration order:

F, D, B, C, A, E, G, H, I, J, K

Candidate list:

D, G, I, H, B, C, E, A, K, J



Comments on best-first search

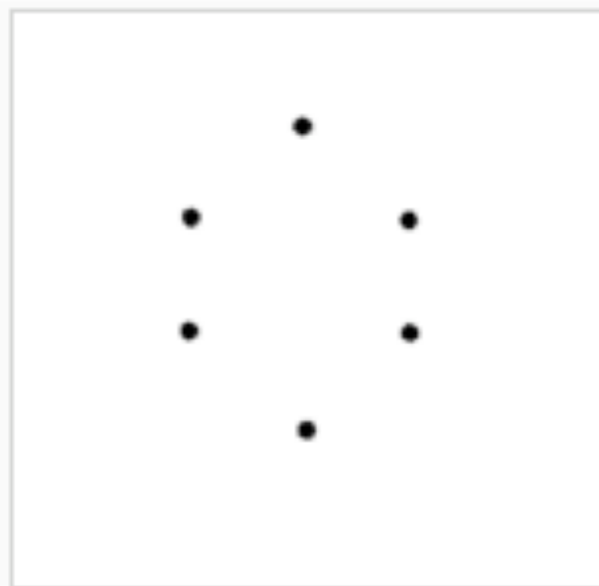
- Best-first search explores graph by according to **priority order**.
E.g. For web-browsing, always explore link with highest PageRank.
IMPORTANT:
 - Need to have a score function, which can be calculated for each node.
 - Need to keep track of candidate nodes.
 - **Pro:** Usually much faster to reach a goal node (e.g. let's say we stop when we reach "A".)
 - **Con:** No advantage if you want to explore the full graph.
-

Graph Topologies

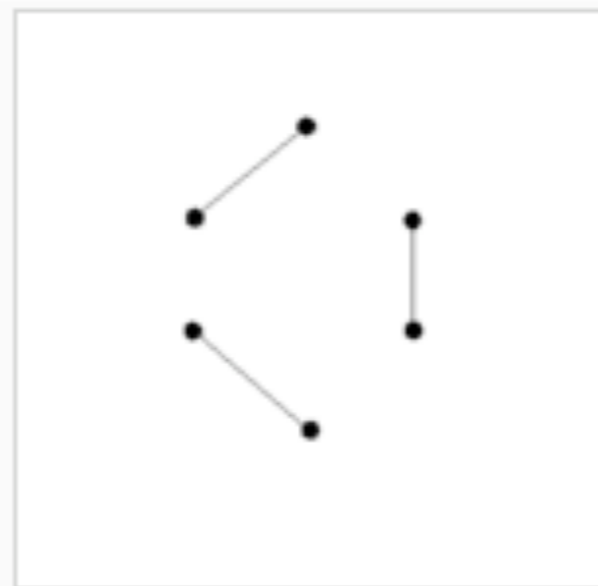
- **Topology** = The arrangement in which the nodes of a graph are connected to each other.
 - Common types of graphs:
 - **Regular graph**
 - **Complete graph**
 - **Random graph**
-

Regular graph

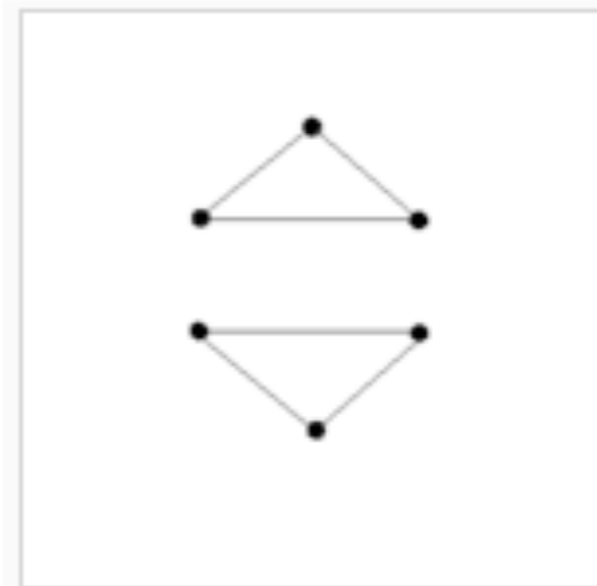
- Main characteristic: Each node has same number of neighbours.



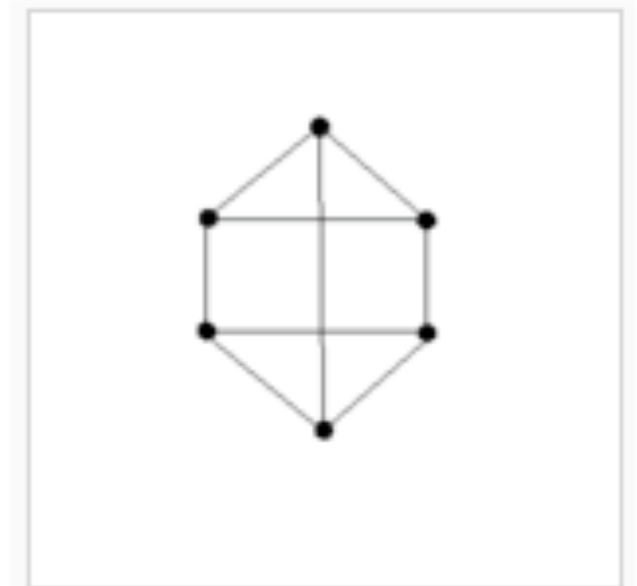
0-regular graph



1-regular graph

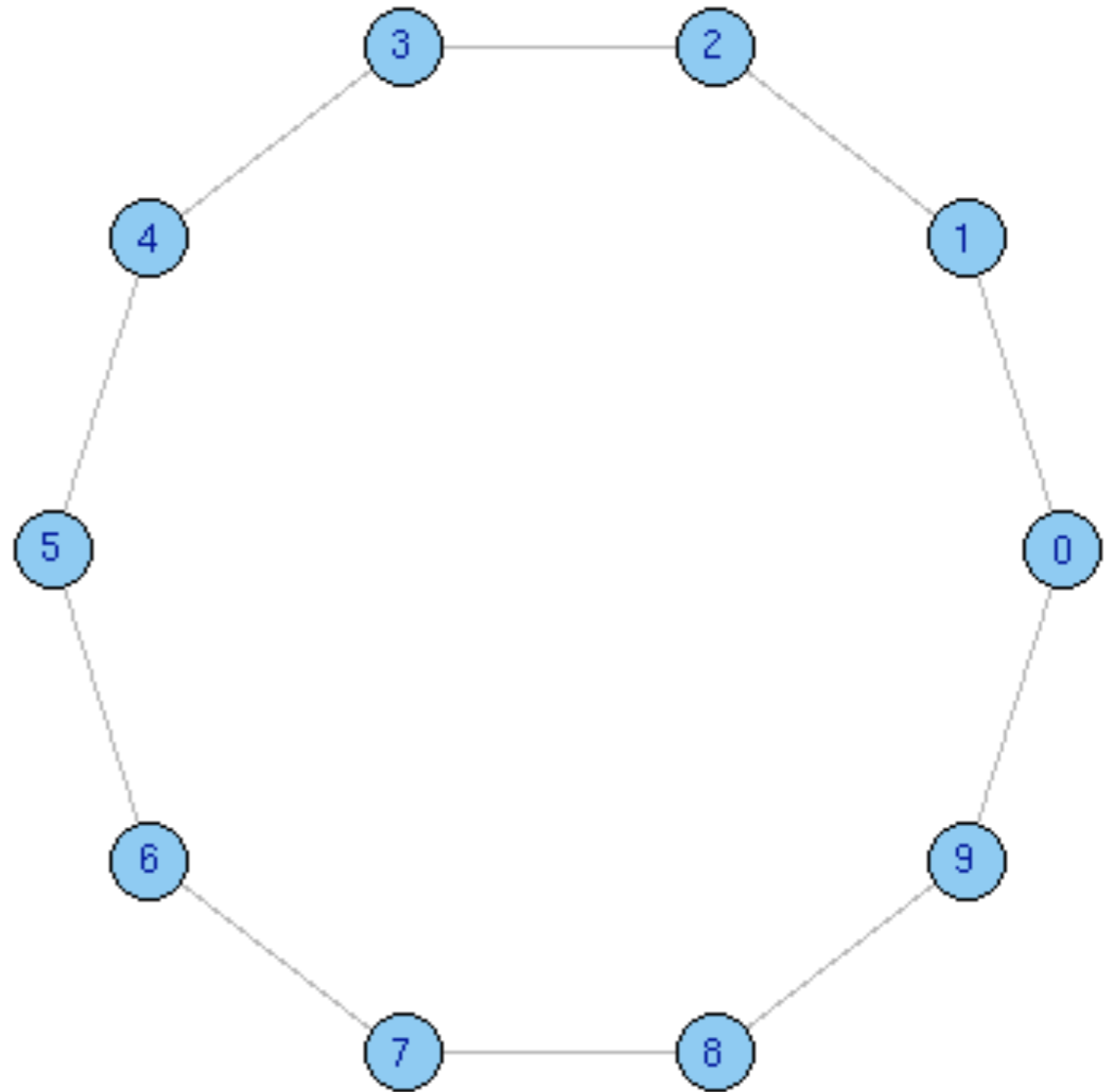


2-regular graph



3-regular graph

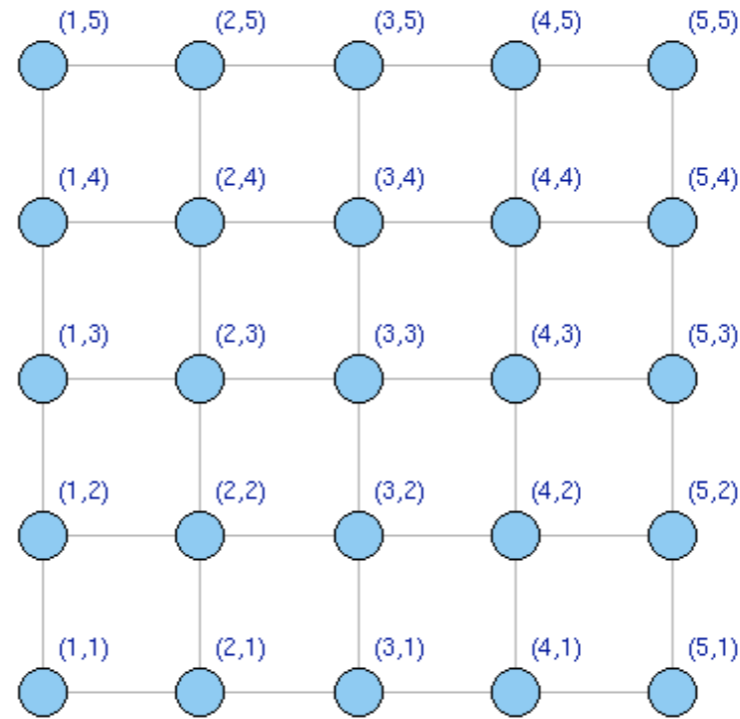
Special regular graph: the Ring



<http://geza.kzoo.edu/~csardi/module/html/>

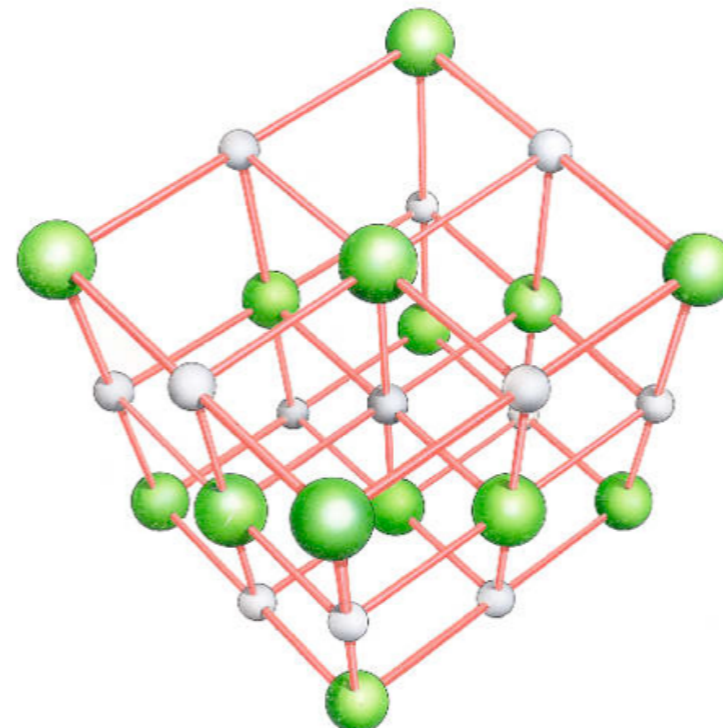
Special regular graph: the Lattice

- This is a common topology to model road networks (in 2-D).



<http://geza.kzoo.edu/~csardi/module/html/>

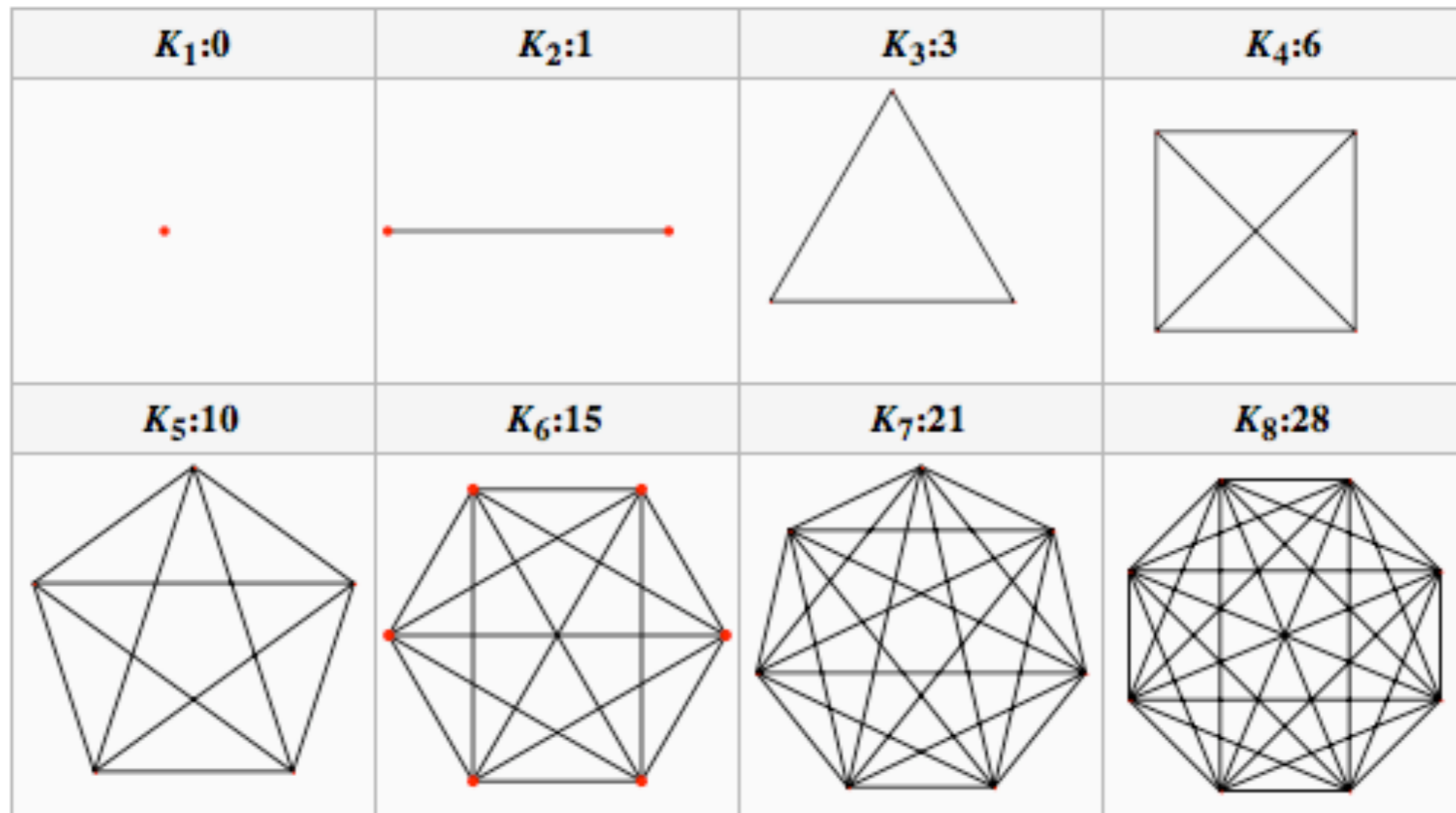
- Also common for molecular diagrams (in 3-D).



<http://www.dkimages.com>

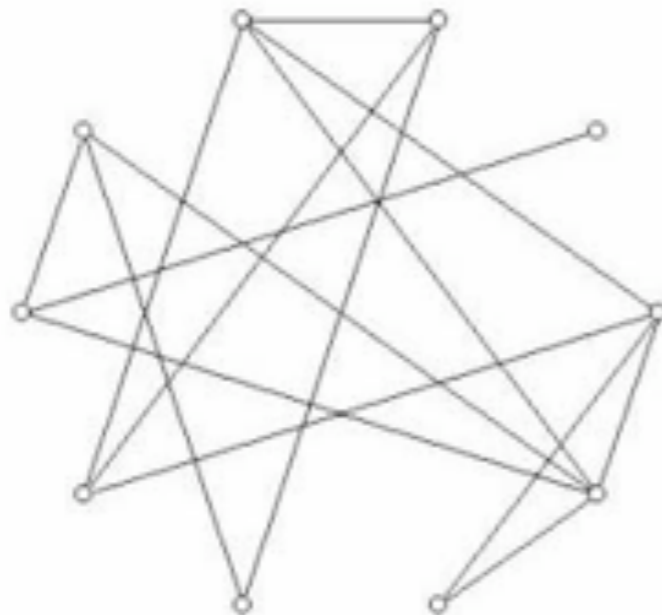
Complete graph (also a regular graph)

- Main characteristic: All pairs of nodes are connected by an edge.



Random graph

- Basic construction: Start with a set of nodes. With probability p , randomly add an edge between any pair of nodes.
- Graph is denoted $G(n,p)$, where n is the number of nodes and p is the probability of a pairwise connection.



Graphs in the real world

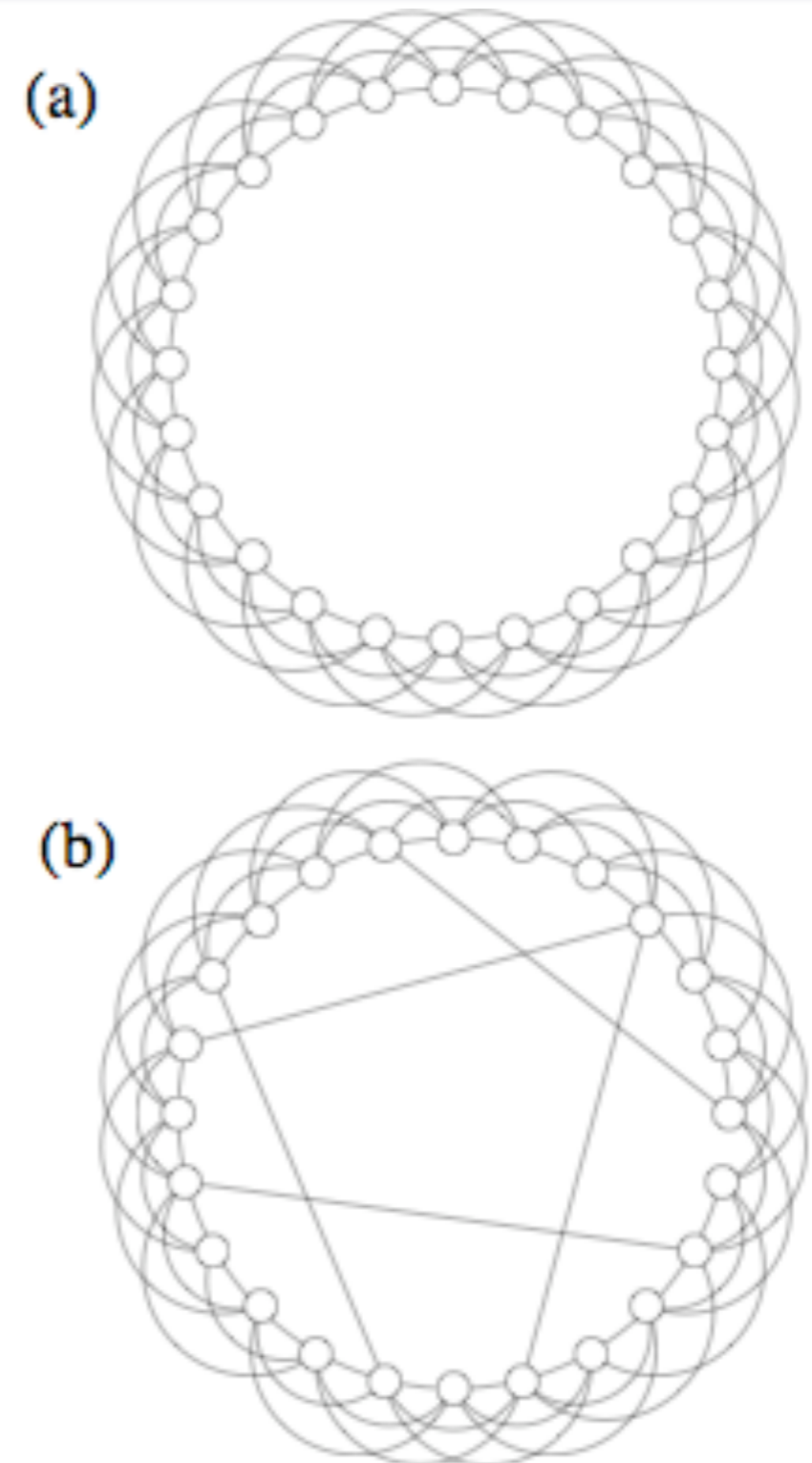
- Think back to our examples:
 - Montreal metro system.
 - Friendship networks.
 - Roadmap of a city.
 - The internet.
 - A maze.
 - Most biological, technological and social graphs/networks are not exactly regular, complete or random.
-

Take-home message

- Main searching algorithms for graphs: Breadth-first search, Depth-first search, Best-first search.
 - Know the steps of each algorithm, and the pros/cons for each.
- Characteristics of the basic types of graphs (regular, complete, random).

Small-world networks

- A **small-world network** is a mix of a regular graph and a random graph.
- Simple construction:
 - Start with a ring made of n nodes and k edges per node.
 - Wire the k edges as for a regular graph.
 - With probability p , re-wire each edge to another random node.



Characteristics of a small-world network

- Key parameters:
 - n controls the size of the graph (= number of nodes)
 - k controls the degree of connectedness (e.g. if $k=n$ then we have a complete graph.
 - p controls the trade-off between “regular” ($p=0$) and “random” ($p=1$)

This correctly models many real-life networks!

Example: Model the spread of an infectious disease

Consider a population of n individuals, connected according to a given topology.

Basic model:

- On day 1: a number b of individuals are infected.
- On day 2 (and subsequent days): we see the effect of that infection
 - Each infected individual can infect each of its neighbours with probability h .
 - Each infected individual is cured with probability g .

We could complicate this model significantly, e.g.

- Individuals have probability of dying from the disease.
 - Individuals develop immunity so can't get the disease more than once.
 - Individuals take a variable number of days to develop symptoms after contagion.
-

Analysis of the model

Now we can ask lots of interesting questions!

- For what values of infection rate h and remission probability g can we keep infection rate at less than 10% of the population?
- What is the critical base rate b at which the disease infects half of n in less than a week?
- What is the impact of the graph topology on the spread of a disease?
- What is the impact of a specific intervention strategy (e.g. through manipulating h) on the spread of the disease?

How do we get these results?

Simulating a graph

- Need to simulate our graph, to capture the change of state in the infected population.
 - What do you remember about finite-state machines?
 - States + Transition graph. Use this here!
 - Pick values for n , b , g , and h .
 - The state is described by a separate variable, $n_i = \{\textit{healthy}, \textit{infected}\}$ for each node.
 - The transition graph expresses the effect of the infection.
-

Scratch simulation

infection:)



<http://scratch.mit.edu/projects/zevbo/1372318>