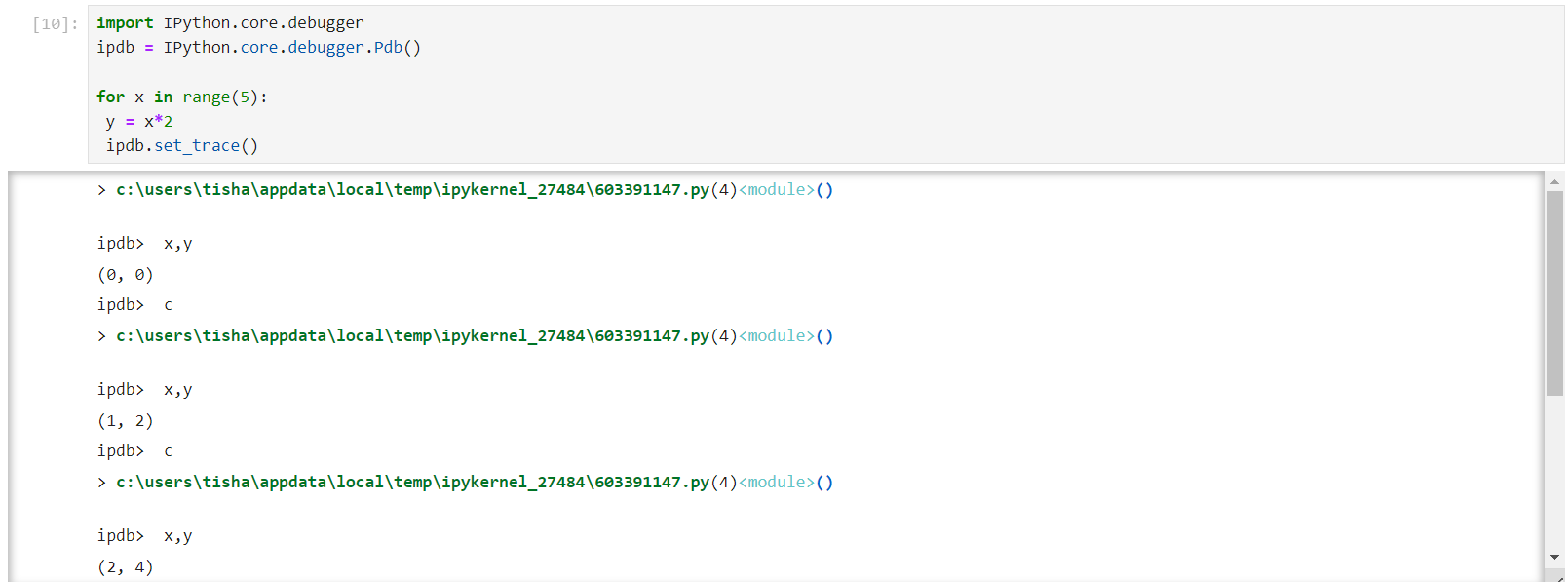
**ADVANCED ALGORITHMS AND COMBINATORIAL OPTIMIZATION**

KAMALDEEP SINGH | ksoberoi@cesi.fr

**NOTES A**

**DEBUGGER’S INTERNAL COMMANDS**

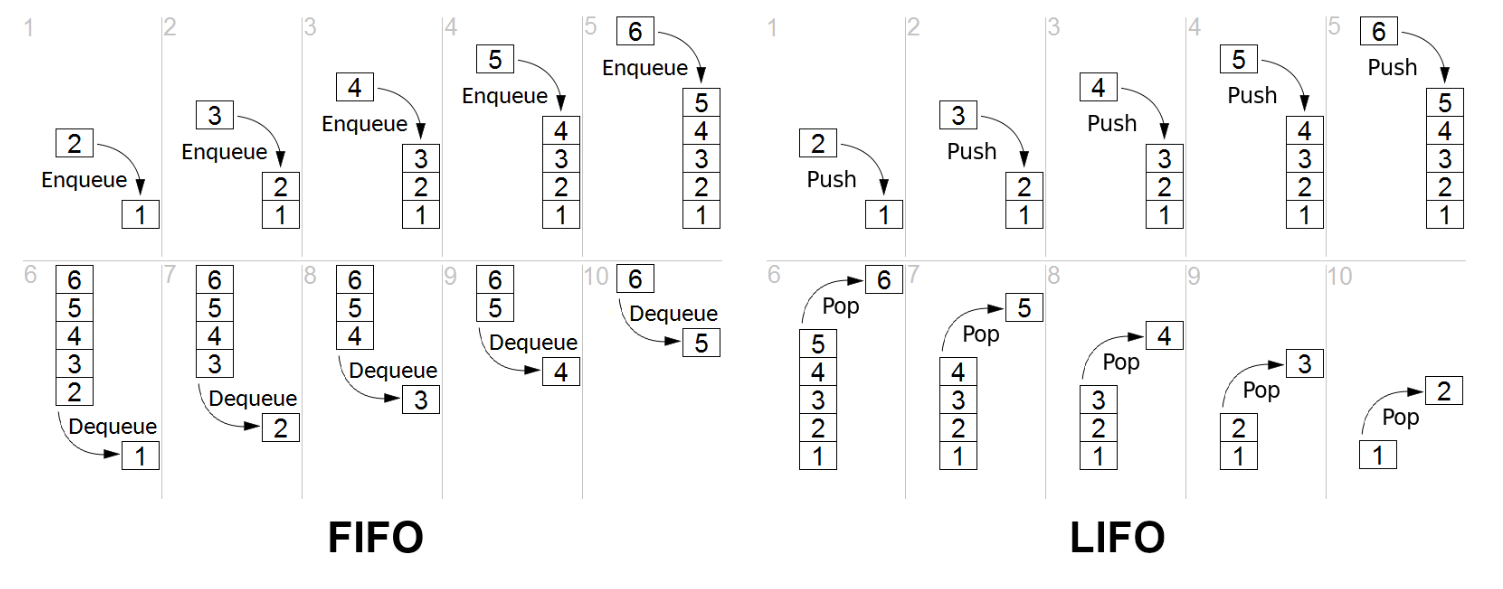
* n – skips the next line
* q – quits the debugger
* c – continues running the program until the end
* s – enters the function of the next line



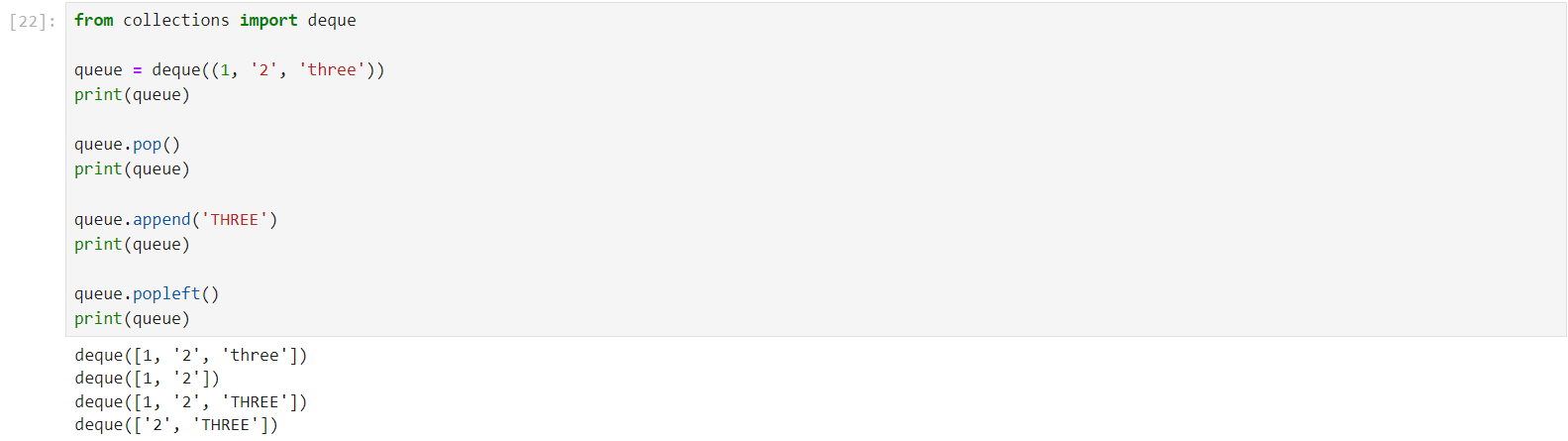
* %%timeit # measures the average running time (with standard deviation) of the cell

**DATA STRUCTURES**

* Containers – an object that contains other objects. For eg – [], (), {}
  + Nested containers may not have the same size and that is okay.
  + Containers are references (pointers) to objects.
  + Shallow Copy – constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
    - copy.copy(obj) - Return a shallow copy of obj.
    - copy.replace(obj, /, \*\*changes) - Creates a new object of the same type as obj, replacing fields with values from changes.
  + Deep Copy – constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.
    - copy.deepcopy(obj[, memo]) - Return a deep copy of obj.
* Lists - most flexible container in Python. It can be modified, increased, decreased, by index or by value.
  + remove(*value*) => ValueError if value does not exist in the list
  + pop(*index*) => IndexError if index does not exist in the list
  + append(*value*)
  + clear()
  + count(*value*)
  + extend(*list*)
  + index(*value*) => ValueError if value does not exist in the list
  + insert(*index, value*)
  + reverse()
  + sort()
* Tuples – immutable and have significantly lower costs than lists.
  + count(*value*)
  + index(*value*) => ValueError if value does not exist
* Stacks and Queues



* + Collection library
    - deque – double-ended queue
      * addition and deletion are done in constant time.
      * Can be created from any sequency but tuple will be faster

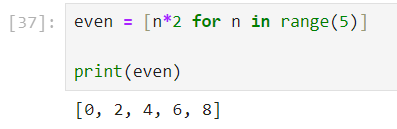


* Dictionary - a collection of items in which each item in a dictionary is a key-value pair.
  + Based on hash-table so accessing an element is O(1)
  + Mutable (like lists)
  + Methods –
    - clear()
    - copy()
    - dict.fromkeys(*key, value*)
    - get(key) => Gives the value, return None if key not in dictionary
    - items() => Returns a list containing a tuple for each key value pair
    - keys() => Returns a list containing the dictionary's keys
    - values() => Returns a list containing the dictionary's values
    - pop(*key*) => KeyError if key doesn’t exist
    - popitem() => Deletes the last key-value pair
    - setdefault(*key, value*) => If the key does not exist, this value becomes the key's value; Default value None
    - update(*{key : value}*)
* Sets - store multiple unique items in a single variable.
  + collection which is unordered, immutable, and unindexed.
  + add(*value*)
  + clear()
  + copy()
  + *z = x*.difference(*y*)
  + *x.*difference\_update(*y*)
  + discard(*value*)
  + *z = x.*intersection(*y*)
  + *x.*intersection\_update(*y*)
  + *z = x.*isdisjoint(*y*)
  + *z = x.*issubset(*y*)
  + *z = x.*issuperset(*y*)
  + pop()
  + remove(*value*) => Different from discard as discard will not raise an error if value does not exist but remove will
  + *z = x.*symmetric\_difference(*y*)
  + *x.*symmetric\_difference\_update(*y*)
  + *z = x.*union(*y*)
  + *x.*update(*y*)

Time taken: set < dictionary < tuple < deque < list

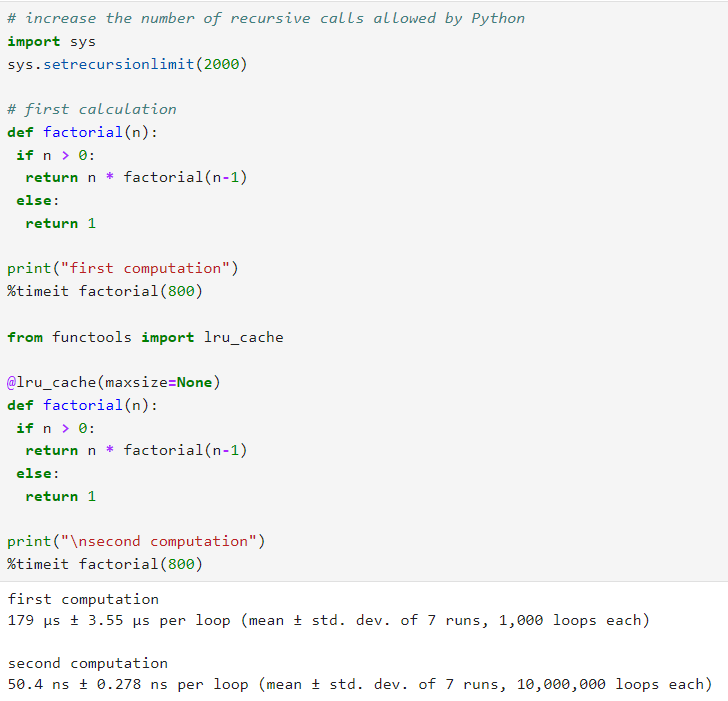
**ENUMERATE()** – gives the index:value pairs (except for dictionaries in which it gives – index:key pairs)

**LIST COMPREHENSION** – offers a concise way to create a new list based on the values of an existing list.



**GENERATORS** – a function that returns an iterator. Basically, *yield* is a kind of return, but it does not interrupt the execution of the function, which makes it possible to return several successive values.

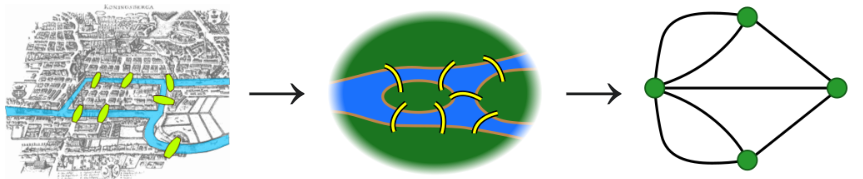
**MEMOIZATION\*** – an optimization technique used to speed up computer programs by caching the results of expensive function calls and returning them when the same inputs are encountered again. Requirements – the function must be deterministic and it must have no side-effects, i.e. the function must be *pure*.



*Referential transparency* is a property of a function that allows it to be replaced by its equivalent output. In simpler terms, if you call the function a second time with the same arguments, you're guaranteed to get the same returning value.

Using *cache* and *lru\_cache* are exactly the same [Source: [stackoverflow](https://stackoverflow.com/questions/70301475/difference-between-functools-cache-and-lru-cache#:~:text=So%2C%20in%20short%3A%20cache%20and,doesn't%20make%20much%20sense.)].

**KONIGSBERG SEVEN BRIDGES**

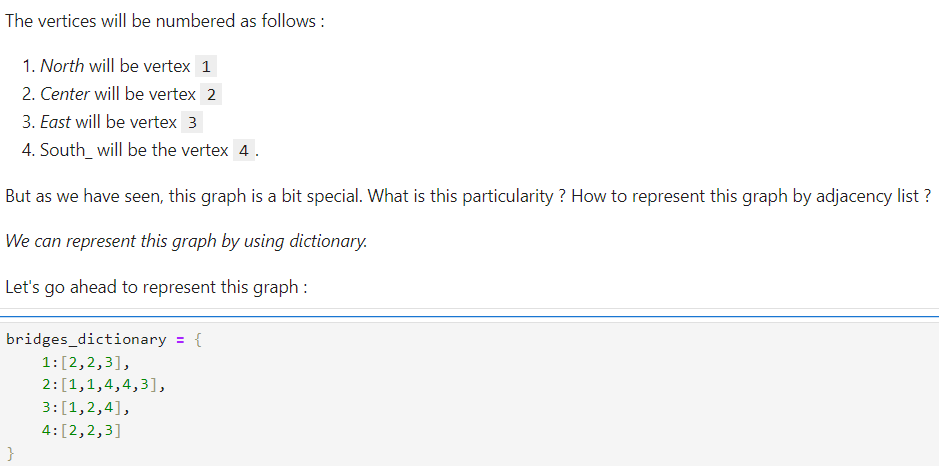
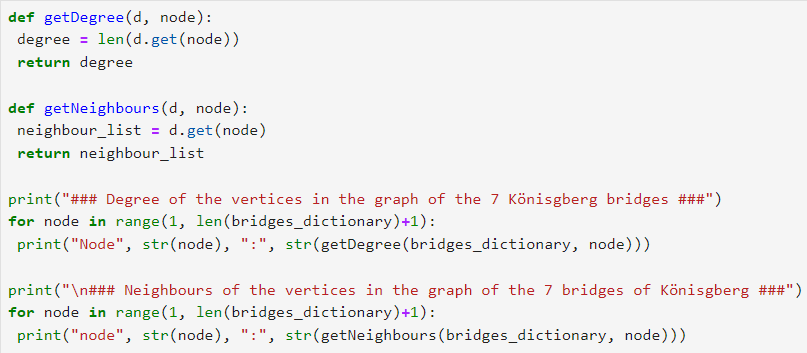
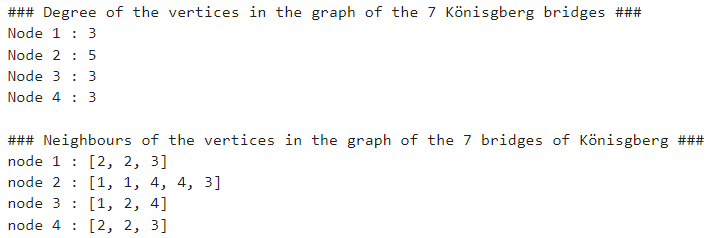


The problem is determining whether or not there is a walking route through the streets of Königsberg that allows, from any given starting point, to cross each bridge exactly once, and to return to its starting point.

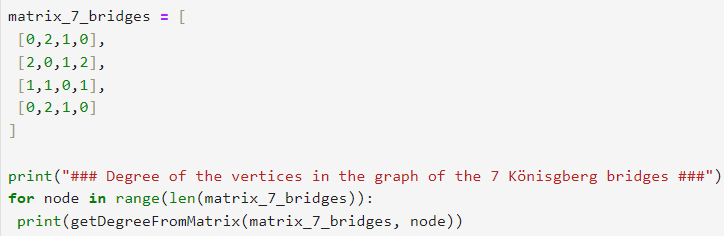
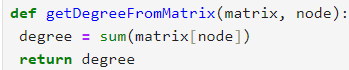
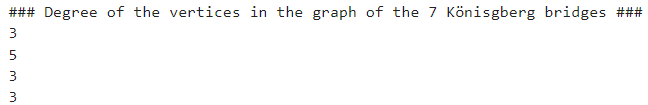
Constraints:

* can’t reach any place other than the bridges
* can’t access a bridge without crossing to its other end

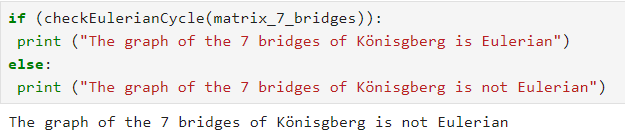
Dictionary [adjacency list]

2-D List [Matrix]

Checking if it is a Eulerian Cycle:

You can turn it into a Eulerian Path if you remove one of the bridges (check notes in diary).

**EULER PATH AND CIRCUIT**

In graph theory, a Eulerian trail (or Eulerian path) is a trail in a finite graph that visits every edge exactly once (allowing for revisiting vertices). Similarly, a Eulerian circuit or Eulerian cycle is a Eulerian trail that starts and ends on the same vertex.

Euler's Theorem: A connected graph has a Euler cycle if and only if every vertex has even degree.

For the existence of Eulerian trails, it is necessary that zero or two vertices have an odd degree.

**HAMILTONIAN PATH AND CYCLE**

A Hamiltonian path (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once.

A Hamiltonian cycle (or Hamiltonian circuit) is a cycle that visits each vertex exactly once.

A Hamiltonian path that starts and ends at adjacent vertices can be completed by adding one more edge to form a Hamiltonian cycle, and removing any edge from a Hamiltonian cycle produces a Hamiltonian path.

A graph is Hamiltonian if it has a Hamilton cycle, and semi-Hamiltonian if it is not Hamiltonian but does have a Hamilton path.

The computational problems of determining whether such paths and cycles exist in graphs are NP-complete.

**GRAPH THEORY**

* Graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects.
* A graph in this context is made up of vertices (also called nodes or points) which are connected by edges (also called arcs, links or lines).
* A distinction is made between undirected graphs, where edges link two vertices symmetrically, and directed graphs, where edges link two vertices asymmetrically.
* For each vertex v, the set of vertices which are adjacent to v is called the neighbourhood of v.
* The degree of a vertex v, written d(v), is the number of ends of edges which connect to that vertex.
* Euler’s handshaking lemma: The sum of the degrees of the vertices of a graph is equal to twice the number of edges.
* Empty graphs: En has n vertices and no edges.
* Complete graphs: Kn has n vertices and each vertex is connected to each other vertex by precisely one edge.
* A walk: is a sequence of the form v1, e1, v2, e2 . . . vr, for some r > 1, where the vi are vertices and ei is an edge from vi to vi+1 for each 1 ≤ i < r.
* A trail: is a walk in which no edge appears more than once.
* A path: is a walk in which no vertex appears more than once.
* A graph G is connected if and only if there is a path between every pair of vertices.
* A tree: a connected graph with no cycles.
  + A tree must be a simple graph.
* A forest: a graph with no cycles.
* A connected graph with n vertices has at least n − 1 edges.

**KRUSKAL’S ALGORITHM**

* A weighted graph is a graph where each edge has a positive number (or “weight”) associated with it.
* The weights typically represent distance, time or cost of travel between vertices.
* The weight of a subgraph is the sum of the weights of all the edges which are included.
* Kruskal’s algorithm finds a minimum-weight spanning tree in a weighted graph.
* Algorithm:
  + list the edges in increasing order of length;
  + consider the smallest remaining edge;
  + if we can add that edge to T without creating a cycle, do so, otherwise discard it;
  + if we have n − 1 edges, T is a spanning tree so stop;
  + otherwise go back to step 2.

**EULERIAN GRAPHS**

* A graph is Eulerian if it has a closed trail which includes every edge. [Eulerian cycle]
* A graph is semi-Eulerian if has a trail which is not closed but which includes every edge. [Eulerian path]

**FLEURY’S ALGORITHM**

* Fleury’s algorithm finds an Euler trail in an Eulerian graph, or a trail which uses every edge in a semi-Eulerian graph.
* A bridge in a connected graph is an edge whose removal will disconnect the graph.
* Algorithm [Eulerian cycle]:
  + Start at any vertex
  + move along any edge
  + delete that edge once you have crossed it
  + only cross a bridge if there is no alternative.
* Algorithm [Eulerian path]
  + Start at either vertex of odd degree
  + move along any edge
  + delete that edge once you have crossed it
  + only cross a bridge if there is no alternative.

**CHINESE POSTMAN PROBLEM**

A postman delivers letters to a set of streets which form the edges of a connected graph G; the vertices of G are junctions between streets. He knows the length of each street (different streets may have different lengths). He needs to start and finish at the same junction, and travel along a walk which covers every street at least once. What is the shortest walk he can take which achieves this?

*So, this is an optimization problem*

* The postman can certainly do no better than the total length L of all the streets, and a route of this length is precisely a Euler trail so he can do this if and only if G is Eulerian. If it is not Eulerian, he will need to travel along some streets more than once.
* If the graph is semi-Eulerian, x and y are the two vertices of odd degree, and the shortest path between x and y has length P then the shortest postman route has length L + P.

Travelling Salesman Problem

A travelling salesman needs to visit each of a group of cities, returning to his starting point. The distance from each city to each other is known, and he wishes to minimise his total distance.

Algorithm for upper bound:

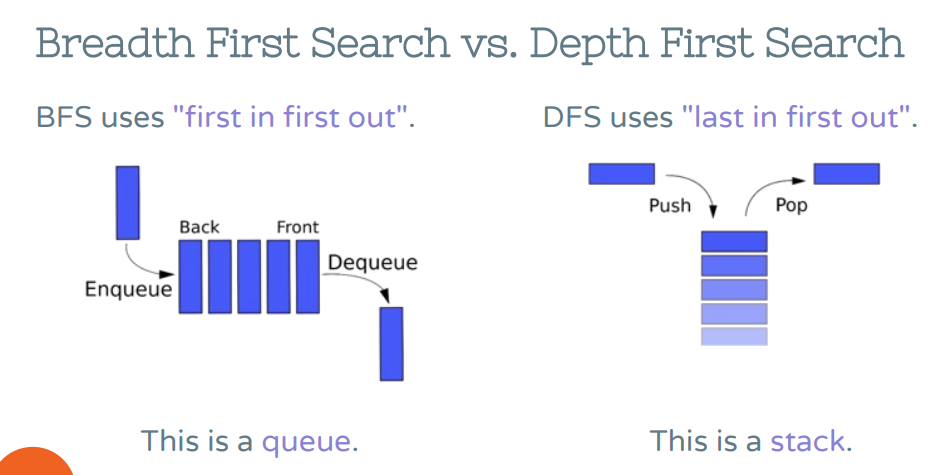
* List the edges in increasing order of weight.
* Start with a single vertex, A, say, and find the shortest edge from A to another vertex.
* Start from the cycle of length 2 which goes along that edge and back again.
* Find the shortest edge from a vertex already in the cycle to a vertex not yet in the cycle.
* Add that edge to the cycle by inserting the new vertex after the vertex which was already in the cycle.
* Repeat steps 4 and 5 until the cycle goes through all the vertices.

Algorithm for lower bound:

* Pick a vertex, say A, and remove it.
* Use Kruskal’s algorithm to find a minimum spanning tree for the remainder.
* Find the two shortest edges from A to the rest of the graph.
* Add these two lengths to the total length of the spanning tree to get a lower bound.

**DEPTH FIRST SEARCH**

* Choose a vertex to start from, A, say.
* Choose any unvisited vertex reachable from A, and go there.
* Keep going, at every stage choosing an unvisited vertex which we are allowed to move to – mark any vertex from which we have only one option.
* Keep going until you either find a Hamilton path or are stuck. – w
* When you are stuck backtrack to the previous vertex where there was a choice and choose differently.



**BREADTH FIRST SEARCH**

* We aim to build up a tree of possible routes.
* Start with A in the first column, then put the possible vertices you can get to from a in the next column
* Then the possible ways to continue these paths in the third column, and so on.
* We will find all possibilities this way.

**DIJKSTRA’S ALGORITHM**

* Dijkstra’s algorithm finds the shortest distance to every vertex.
* Start by marking the start vertex as distance 0, and circle it.
* For each vertex adjacent to the start vertex, we calculate a bound which is the length of the edge to that vertex.
* Find the vertex with the smallest bound among vertices which do not have final answers circled. Circle that bound, and mark the edge which gave that bound.
* For each edge leaving the vertex you have just marked with a final answer, add that answer to the length of an edge to get a bound for the vertex that edge goes to. If it is smaller than the current bound at that vertex, replace the old bound with this one.
* Repeat steps 3 and 4 until all vertices have their exact distances marked.

*Minimum-Cost Maximum Flows is interesting and important for our project (potentially)*

**VERTEX COLOURING**

* Let G be any graph. A vertex colouring of G is an assignment of a colour to each vertex such that every edge goes between two different colours.
* If G has a loop then it has no colourings at all, and it makes no difference for colouring purposes whether we have one or two (or more) edges between x and y.
* The chromatic number of G, written χ(G), is the smallest k such that G is k-colourable.
* Greedy Algorithm:
  + Let G be a simple graph. Choose an ordering of the vertices of G.
  + We will colour the vertices one at a time in order with colours from the set {1, 2, 3, . . .}.
  + When colouring each vertex, we choose the smallest colour which has not already been assigned to one of its neighbours.
  + The worst-case performance of the greedy algorithm can be very bad: for each n there is a bipartite graph on 2n vertices and an ordering for which greedy uses n + 1 colours.

**EDGE COLOURING**

* Let G be a graph with no loops (but multiple edges are permitted). An edge colouring of G is as assignment of a colour to each edge such that any two edges which have a common vertex are assigned different colours.
* We say G is k-edge-colourable if there is an edge colouring of G using at most k colours and define the chromatic index of G, which we write χ’(G), to be the smallest k for which G is k-edge-colourable.

Notes - <https://ptwiddle.github.io/MAS341-Graph-Theory-2017/OldLectureNotes.pdf>

**SUBOPTIMAL**

The suboptimal solution is a feasible solution available at the final iteration (modified, if necessary, to satisfy any hard constraints on the manipulated variables). To determine whether the suboptimal solution provides acceptable control performance for your application, run simulations across your operating range.

Source - <https://fr.mathworks.com/help/mpc/ug/simulate-mpc-controller-using-suboptimal-solution.html>

**HEURISTICS**

* A heuristic is an approximate measure of how close you are to the target.
* It guides you in the right direction.
* It should be easy to compute.

Source: <https://cs.stanford.edu/people/abisee/gs.pdf>

**META-HEURISTICS**

**OPERATION RESEARCH**

* Operations Research (OR) is a discipline of problem-solving and decision-making. It uses advanced analytical methods to help management run an effective organization. Problems are broken down, analyzed and solved in steps.
* Operations Research, also known as management sciences, uses scientific methods to study systems that require human decision-making.
* Consequently, OR helps make the most effective systems design and operation decisions.
* Moreover, OR’s strength and versatility come from its diagnostic power through observation and modelling and its prescriptive power through analysis and synthesis.

**WORKSHOP NOTES**

**NETWORKX**

**ADJACENCY LIST**

**ADJACENCY MATRIX**

**WEIGHTED ADJACENCY MATRIX**

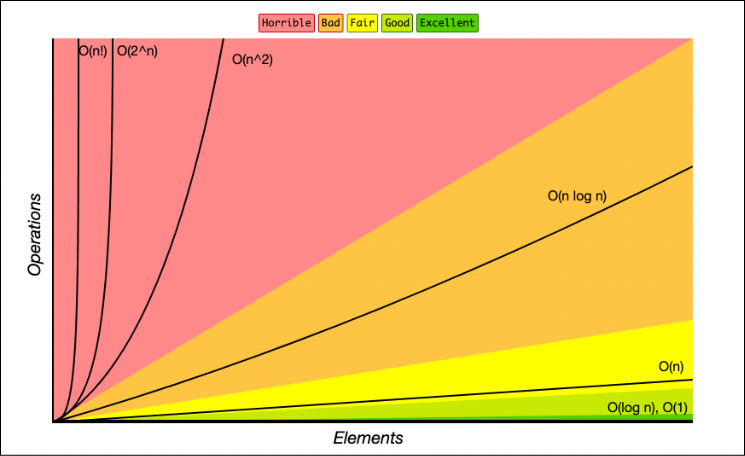
**INCIDENCE MATRIX**

**MULTIGRAPH**

**PSEUDOGRAPH**

**BACKTRACKING ALGORITHM**

**COMPUTATIONAL COMPLEXITY (BIG O NOTATION, GRAPHS, NP)**



**PROBLEM**

* A problem is simply a formal task to be performed, which can be described purely in terms of inputs and matching outputs, without any constraints on the method of the solution.
* Problems can be viewed as functions in the mathematical sense, since they deterministically map an input in a domain of values to a well-defined output in a range of values.
* Different instances (function parameters/problem inputs) of the same problem might generate the same output, but any problem instance (fixed inputs) must always result in the same output every time the function is computed using that particular input.

**DECISION PROBLEM**

* A decision problem is a computational task that decides (deterministically) if its input possesses a certain property and accordingly outputs either “true” or “false”.
* In other words, in the terminology from above, a decision problem is a problem with a “true” or “false” output.
* As a function, its output range is the set of values “true” and “false”.
* P and NP are classes of decision problems.

**OPTIMIZATION PROBLEM**

* Optimization problems are concerned with finding the best answer to a particular input.
* Optimization problems arise naturally in many applications, such as the traveling salesman problem and many questions in linear programming.
* Function and optimization problems are often transformed into decision problems by considering the question of whether the output is equal to or less than or equal to a given value.
* This allows the complexity of the corresponding decision problem to be studied; and in many cases the original function or optimization problem can be solved by solving its corresponding decision problem.

**DETERMINISTIC TURING MACHINE**

* Turing machines are a model of computation. It is believed that anything that can be computed can be computed by a Turing Machine.
* A Turing Machine is a tuple (Q, Σ, δ, s, h) where
  + Q is a finite set of states. It has the states s, qacc, qrej.
  + Σ is a finite alphabet. It contains the symbol #.
  + δ: Q − {qacc, qrej} × Σ → Q × Σ ∪ {R, L}
  + s ∈ Q is the start state, qacc is the accept state, qrej is the reject state.
* *P = DTIME(nO(1))*
* *NP = NTIME(nO(1))*

**ASYMPTOTIC COMPLEXITY [P / NP]**

* An algorithm is said to be efficient if it runs in polynomial-time.
* The class P comprises the decision problems that are decidable and for which an efficient algorithm exists to solve them. Since P only deals with decision problems, not all problems that can be solved in polynomial time are P.
* The class NP comprises decision problems that are decidable and for which an efficient yes-verifier exists.
* The class of decidable problems with such verifiers for the “no”-answers is called co-NP.

**POLYNOMIAL TIME ALGORITHM**

* An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is O(nk) for some nonnegative integer k, where n is the complexity of the input.
* Polynomial-time algorithms are said to be "fast."
* Most familiar mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, powers, and logarithms, can be performed in polynomial time.
* Computing the digits of most interesting mathematical constants, including π and e, can also be done in polynomial time.

**CERTIFICATE ALGORITHM**

* Certificate algorithms are cryptographic algorithms that describe the mathematical procedures that are used for creating key pairs and performing digital signature operations.

**VEHICLE ROUTING PROBLEM**

**TRAVELLING SALESMAN PROBLEM**

**3-SAT PROBLEM [P AND NP]**

**SIMPLEX**

**SYSTEMS OF EQUATIONS/INEQUATIONS**

**DYNAMIC PROGRAMMING**

**LINEAR PROGRAMMING**

**METAHEURISTIC**

**ADAPTIVE APPROACH**