Design Automation Renegades

Globetrotting Division

Boilerplate Code: Data Structures and Algorithms for Design Automation

Zhiyang Ong [1]

Report on
Common Data Structures and Algorithms
Found in Bolierplate Code for
Design Automation Software

July 11, 2019

[1]Email correspondence to: ongz@acm.org

Abstract

This report describes the design and implementation of common data structures and algorithms, as well as "computational engines" that are found in electronic design automation (EDA) software.

Data structures and algorithms for digital VLSI and cyber-physical system design include: binary decision diagrams (BDDs), AND-inverter graphs (AIGs), and their associated algorithms for optimization, traversal, and other operations (such as graph matching). Common computational engines for digital systems would include: optimization and verification engines for deterministic and nondeterministic finite state machines; decision procedures for the boolean satisfiability problem (SAT solvers) and satisfiability modulo theories (SMT solvers); quantified boolean formula (QBF) solvers; and SAT and SMT solvers for maximum satisfiability (i.e., Max-SAT and Max-SMT solvers).

Regarding EDA problems that require numerical computation (in digital, analog, or mixed-signal VLSI design), the data structures and algorithms for circuit simulation based on sparse graph would be required. In addition, techniques for model order reduction shall be implemented.

Computational engines for statistical and probabilistic analyses or stochastic modeling can include data structures and algorithms for partially observable Markov decision processes (POMDPs) and Markov chains. Tools for analyses of queueing systems (based on queueing theory) should be included.

Regarding cyber-physical systems and mixed-signal circuits, hybrid automata can be used to represent these circuits and systems.

Optimization engines for EDA include: solvers for different types of mathematical programming, such as linear programming (LP), integer linear programming (ILP), mixed-integer linear programming (MILP), quadratic programming (QP), convex programming (CP), geometric programming (GP), and second-order conic programming (SOCP); solvers for pseudo-boolean optimization (PBO solvers) and weighted-boolean optimization (WBO); and meta-heuristics (e.g., evolutionary algorithms, simulated annealing, and ant colony optimization).

Algorithms shall be implemented using parallel programming, in a scalable style. In addition, considerations shall be given to the use of constraint programming.

More stuff to be included...

Revision History

Revision History:

1. Version 0.1, December 23, 2014. Initial copy of the report.
2. Version 0.1.1, September 16, 2015. Added sections for mathematics and statistics, and the abstract.
3. Version 0.1.2, November 10, 2018. Added sections for graphs, including directed graphs (digraphs), directed acyclic graphs (DAGs), and undirected graphs.

Contents

Chapter 1

Algorithms

This section documents algorithms that I have implemented for my C++ -based boilerplate code repository.

A template for typesetting algorithms is shown in Procedure 1.

NAME OF THE ALGORITHM(*ARGUMENTS*)
    // *Input ARGUMENT #1: Definition1*
    // *Input ARGUMENT #2: Definition2*
1   BODY OF THE PROCEDURE
    // *A while loop.*
2   **while** [condition]
3   [Something]
    // *A for loop.*
4   **for** *Var* = [initial value] **to** [final value]
5   [Something]
    // *An if-elseif-else block.*
6   **if** [Condition1]
7   Blah...
8   **elseif** [Condition2]
9   Blah...
10 **elseif** [Condition3]
11   Blah...
12 **else**
13   Blah...
    // *A variable assignment.*
14 *blah* = *A*[*j*]
        // *This is indented with a tab.*
    // *What is the output of this procedure?*
15 **return**

1.1 Notes on Algorithm Analysis and Design

[45, §A.2] shows you how to manipulate and bound summations, so that we can obtain an order of (computational) complexity for summations. Specifically, [45, §A.2, pp. 1154–1156] shows us how

to approximate the bound of summations by using finite integrals. *Are these related to recurrence relations???*

1.2 Resources for Algorithms

Resources for algorithms:

1. Collected Algorithms (CALGO): http://calgo.acm.org/
2. Netlib Repository at UTK and ORNL [69]: http://www.netlib.org/
3. "The Stony Brook Algorithm Repository" by Steven Skiena [193]: http://algorist.com/algorist.html
4. Cosmos (from OpenGenus Foundation): https://github.com/OpenGenus/cosmos
5. *Wikipedia*:
   (a) https://en.wikipedia.org/wiki/List_of_algorithm_general_topics
   (b) https://en.wikipedia.org/wiki/List_of_algorithms#Graph_algorithms

Chapter 2

Data Structures

2.1 Basic Data Structures

"[A **list** is a] container of variable length , and [a **tuple** is a] container [of] fixed length" [205, §4.3 pp. 111].

A list (in *Prolog*) can be deconstructed into [*Head* | *Tail*], where *Head* refers to the first element of the list and *Tail* refers to the rest of the list; on the other hand, tuples cannot be similarly deconstructed [205, §4.3 pp. 113].

See [85, §4, pp. 48] for a figure to describe an ontology of data structures, except graphs.

2.2 Tree Data Structures

From https://en.wikipedia.org/wiki/Tree_(data_structure), a tree data structure can be represented by a directed acyclic graph, or undirected acyclic graph. The choice of a DAG versus an undirected acyclic graph depends on how the designer(s) want to represent the relationship between a given parent node and a given child node in the tree.

2.3 Graph Data Structures

2.3.1 Graph Theory

A **graph** G is an ordered pair, $G = (V, E)$, of a vertex/node set V and an edge set E [226].

We denote the number of vertices, or the cardinality of the vertex/node set, as $n = |V|$. Likewise, we denote the number of edges, or the cardinality of the edge set, as $m = |E|$ [85, §52.2, pp. 845].

Types of **finite graphs** [226]:

1. **undirected graph** [226]:
   (a) **simple graph**, or **undirected simple finite graph** [226]:
      i. Does not allow **multiple edges** (or, **parallel edges**, or **multi-edges**) between any pair of vertices/nodes in the graph, nor **(self-)loops**.
      ii. A **symmetric loopless directed graph** is a graph that has an edge (v_2, v_1) for each edge (v_1, v_2), and it does not contain any self-loops (i.e., edges $v_i, v_i, \forall i \in V$).

iii. Therefore, the edges of a simple graph form a set, as opposed to multigraphs that have multisets of edges.
iv. An edge is a two-element subset of V; other graphs (i.e., hypergraphs) can have edges with more than two nodes.

(b) **multigraph** [231]:
i. A **multigraph** allows **multiple edges** (or, **parallel edges**, or **multi-edges**) to exist between any pair of vertices (or, nodes) in the graph. Alternatively, for any pair of vertices (or, nodes) in the multigraph, they allow multiple edges to be incident to them [231]. [87, §13.1, pp. 596] uses a **collection** to represent the **group of edges**. I prefer the term **multiset**, **bag**, or **mset**; see https://en.wikipedia.org/wiki/Multiset.
ii. **pseudograph** [231]:
- Some authors/people use pseudographs and multigraphs interchangeably/synonymously.
- Other people use pseudographs to refer to multigraphs that allow self-loops [45, §B.4, pp. 1168] (or loops) [218].

iii. A **planar graph** remains planar, or preserves its planarity, when its edges become multiple edges by the addition of edges to edges in the original graph Reference this!!!.:
- Does the following reference suffice **for all conditions**? http://jgaa.info/accepted/2004/BoyerMyrvold2004.8.3.pdf.

iv. Representations for different types of **multigraphs** [231]:
- A **multidigraph** is also known as **quiver** [234].
- **undirected multigraph, which edges have no identity**, is an ordered pair (or 2-tuple) $G = (V, E)$, where V is a set of vertices (or, nodes), E is a multi-set of undirected edges (or unordered pairs of vertices).
- **undirected multigraph, which edges have an identity each (or, where each edge has an identity)**, is an ordered triple (or 3-tuple) $G = (V, E, r)$, such that $r : E \to \{\{x, y\} : x, y \in V\}$ is a function that assigns each edge to an unordered pair of vertices (i.e., endpoint nodes).
- **directed multigraph (or multidigraph or quiver), which edges have no identity**, is an ordered pair (or 2-tuple) $G = (V, E)$. Here, V is a set of vertices (or, nodes), and E is a multi-set of ordered pairs of vertices (i.e., directed edges, directed arcs, or arrows)
- A **directed multigraph (or multidigraph or quiver), which edges have an identity each (or, where each edge has an identity)**, is a 4-tuple $G = (V, E, s, t)$. Here, V is a set of vertices (or nodes), E is a multi-set of ordered pairs of vertices (i.e., directed edges, directed arcs, or arrows), $s : E \to V$ so that each edge is assigned to its source node(s), and $t : E \to V$ so that each edge is assigned to its destination/target node(s).
- A **mixed multigraph** is a(n) (ordered) 3-tuple $G = (V, E, A)$, where V is a set of vertices (or, nodes), E is a set of undirected edges, and A is a multi-set of directed edges/arcs.

v. A **labeled multigraph** is a 6-tuple $G = (\sum_V, \sum_E, V, E, l_V, l_E)$. Here, V is a set of vertices (or, nodes), and E is a multi-set of ordered pairs of vertices (i.e., directed edges, directed arcs, or arrows), $\sum_V$ is the finite alphabet of available vertex labels, $\sum_E$ is the finite alphabet of available edge labels, $l_V : V \to \sum_V$ is a map describing the labeling of the vertices, and $l_E : E \to \sum_E$ is a map describing the labeling of the edges.
vi. A **labeled multidigraph** is also known as a **labeled, directed multigraph**. It is a

8-tuple $G = (\sum_V, \sum_E, V, E, s, t, l_V, l_E)$. Here, V is a set of vertices (or, nodes), and E is a multi-set of ordered pairs of vertices (i.e., directed edges, directed arcs, or arrows), $\sum_V$ is the finite alphabet of available vertex labels, $\sum_E$ is the finite alphabet of available edge labels, $s : E \to V$ so that each edge is assigned to its source node (or, is a map that assigns each edge to its source node), $t : E \to V$ so that each edge is assigned to its destination/target node (or, is a map that assigns each edge to its destination/target node), $l_V : V \to \sum_V$ is a map describing the labeling of the vertices, and $l_E : E \to \sum_E$ is a map describing the labeling of the edges.

vii. References:
- **Multiple edges**: https://en.wikipedia.org/wiki/Multiple_edges
- **Multigraph** [231]: https://en.wikipedia.org/wiki/Multigraph
- **Graph labeling** [217]:
  - https://en.wikipedia.org/wiki/Graph_labeling
  - "**Graph labeling** is the assignment of labels," which can be represented by numbers and/or strings, to edges and/or vertices of a graph.
  - **Vertex labeling** is a function of V that assigns a set of labels to V; or, it is a function of Vthat assigns a label to each vertex.
  - A **vertex-labeled graph** is a graph with a defined vertex labeling function.
  - **Edge labeling** is a function of E that assigns a set of labels to E; or, it is a function of Ethat assigns a label to each edge.
  - An **edge-labeled graph** is a graph with a defined edge labeling function.
  - A **weighted graph** is an **edge-labeled graph**, such that the edge labels are members of an ordered set (e.g., the set of real numbers $\mathbb{R}$).
  - The term **labeled graph** generally refers to a **vertex-labeled graphs** with unique labels (e.g., $\{1, \ldots, |V|\}$, where $|V|$ is the number of vertices in the graph or the cardinality of V), unless otherwise specified.

(c) **hypergraph**:

i. Related to spectral graph theory, which involves linear algebra.

ii. References for hypergraphs:
- [29]
- [43]
- [6]
- [23]
- [110]
- Not so good references: [19, 35, 37, 51, 125, 130, 178, 185]

(d) **multidimensional networks**:

i. One of the distinguishing features of multi-dimensional edges is multiple edges.

(e) **mixed graph**:

i.

(f) **Planar graph** [233]:

i. https://en.wikipedia.org/wiki/Planar_graph

ii. Are all planar graphs sparse graphs? [233]

iii. "A **planar graph** is a graph that can be embedded in the plane", such that when the graph is visualized in 2-dimensions, its edges only intersect at their endpoints (or, its edges do not intersect each other) [233].

- Such visualizations are known as **planar embeddings of the graph**, or **plane graphs** [233].
- A **planar embedding of a planar graph** $G_P = (V_P, E_P)$ is a mapping of vertices $v_i \in V_P = \{v_1, \ldots, v_n\}, \forall i \in \{1, \ldots, n\}$, to points on a plane, and a mapping of edges $e_j \in E_P = \{e_1, \ldots, e_m\}, \forall j \in \{1, \ldots, m\}$, to plane curves on that plane, such that no planar curve intersects another planar curve and planar curves are connected at points in that plane [233].

(g) **dipole graph**:

i. A **dipole graph** (or **bond graph** – not that kind of **bond graph**) has a set of only two vertices, and a set of (parallel) edges between these vertices.:
   - A **bond graph** is a graphical representation of a physical dynamic system, and represents exchanges of physical energy; see https://en.wikipedia.org/wiki/Bond_graph.

ii. An **order-n dipole graph** Dn is a dipole graph with n edges, and is a dual to the cycle graph C_n.

iii. References:
   - **Multiple edges**: https://en.wikipedia.org/wiki/Multiple_edges
   - **Dipole graph**: https://en.wikipedia.org/wiki/Dipole_graph

(h) A **dual graph** $H = \{V_H, E_H\}$ of a **planar graph** $G = \{V_G, E_G\}$ [233] is a graph that maps each vertex $v_i \in V_G = \{v_1, \ldots, v_n\}, \forall i \in \{1, \ldots, n\}, n \geq 3$, to each face of H, and maps each vertex $v_j \in V_H = \{v_1, \ldots, v_m\}, \forall j \in \{1, \ldots, m\}$, to a face in G, and each edge $e_k \in E_H = \{e_1, \ldots, e_o\}, \forall k \in \{1, \ldots, o\}$, connects adjacent faces of G that are separated by an edge. Hence, each edge $e_l \in E_G = \{e_1, \ldots, e_p\}, \forall l \in \{1, \ldots, p\}$, corresponds to a dual edge e_k in H, such that the endpoints of e_k are the dual vertices corresponding to faces on both sides of e_l [223].

i. The definition of H depends on the **planar embedding** of G, since the **duality property** is a property of the **planar embedding** (as opposed to planar graphs that can be embedded, but its embedding is not known yet) [223].

ii. Hence, a planar graph can have multiple dual graphs, since it can have multiple **planar embeddings** [223].

iii. "The property of being a dual graph is symmetric," such "that if H is the dual of a connected graph G [(the **primal graph**)], then G is [also] a dual of H" [223].

iv. Graph properties and structures of the **primal graph** G have **dual properties and structures** in the dual graph H [223]. Such graph properties and structures include the following [223]:
   - cycles are dual to cuts
   - spanning trees are dual to complements of spanning trees
   - simple graphs are dual to 3-edge-connected graphs

v. Based on the **Jordan curve theorem**, the **dual of the n-cycle graph** (or, **n-vertices graph**) is the **n-edge dipole graph** [223]; see https://en.wikipedia.org/wiki/Jordan_curve_theorem.

vi. "A **plane graph** is said to be **self-dual** if it is **isomorphic** to its dual graph" [223].

vii. Since a **dual graph** H is dependent on the **planar embedding** of the graph G, there does not exist any unique **dual graph** for graph G. That is, a **planar graph** can have **non-isomorphic dual graphs** [223].

(i) A **bouquet grap** B_m "is an undirected graph with one vertex and m**" self-loops** [220]. That is, each edge in the undirected graph B_m is a **self-loop** [218].

2. **directed graph**:

   (a) **directed multigraphs**: see aforementioned notes on "**directed multigraph** (or **multidigraph**), which edges have no identity," and "**directed multigraph** (or **multidigraph**), which edges have an identity each (or, where each edge has an identity)"
   (b) **directed acyclic graphs** (**DAGs**)
   (c) See §2.3.4 for notes on directed graphs (digraphs) and DAGs.

3. **mixed graphs**:
   (a) A mixed graph $G = (V, E, A)$ is a mathematical object that is a 3-tuple (or, triple or triplet), where V is the set of vertices in G, E is the set of undirected edges in G, and A is the set of directed edges in G [229].
   (b) [229] provides definitions for:
      i. directed edges
      ii. undirected edges
      iii. mixed cycle (cycle of a mixed graph)
      iv. acyclic mixed graph
      v. graph coloring problem of mixed graphs:
         A. (strong) proper k-coloring of a mixed graph
         B. weak proper k-coloring of a mixed graph
         C. chromatic number
         D. chromatic polynomial of graph G
         E. weak chromatic polynomial of graph G
   (c) [229] describes applications of mixed graph in the following problems/topics:
      i. Bayesian inference
      ii. Scheduling problem

Additional resources about graphs:

1. **Graph property**, or **graph invariant**:
   (a) https://en.wikipedia.org/wiki/Graph_property
   (b) Notes about vertices/nodes:
      i. If the graph contains an edge connecting vertices u and v, $e = (u, v)$, the vertices u and v are **adjacent** to each other [219].
      ii. The **degree (or valency) of vertex** v, which is denoted by $\delta(v)$ (or, $\deg(v)$ or $\deg v$), is the number of edges that are incident to v [219, 221].:
         - Regarding undirected graphs, the degree of a vertex $\deg(v)$ is equal to the number of edges (i.e., n) [218]; i.e., $\deg(v) = n$.
      iii. An **isolated vertex** is a vertex with degree zero, $\delta(v) = 0$. It is not an endpoint of any edge; or, no edge in a (or, any) graph is defined with an isolated vertex [219, 221].
      iv. A **leaf vertex**, **end vertex**, or **pendent vertex**, is a vertex of degree one, $\delta(v) = 1$, and is an endpoint of only one edge (**pendent edge**) in the graph [219, 221].
      v. A **dominating vertex** is "a vertex with degree $\delta(v) = (n - 1)$ in a graph of n vertices" [221].
      vi. For a vertex v of a directed graph, the **outdegree** of v (denoted by $\delta^+(v)$) is its number of outgoing edges, and the **indegree** of v (denoted by $\delta^-(v)$) is its number of incoming edges. A **source vertex** is a vertex with a zero indegree ($\delta^-(v) = 0$), and a **sink vertex** is a vertex with a zero outdegree ($\delta^+(v) = 0$) [219].
      vii. A vertex v, which has no **incident edges** apart from a(n) (undirected) loop from v to itself, has a degree of two; i.e., $\deg(v) = 2$ [221].
         - "A **loop**, **self-loop** [45, §B.4, pp. 1168], or **buckle**, is an edge that connects a vertex to itself" [218].

- Regarding **undirected graphs**, the **degree of a vertex** with a loop and no other **incident edges** is two; i.e., $\deg(v) = 2$ [218].
- Regarding undirected graphs, a vertex with a loop and no other incident edges has an indegree of one $\delta^-(v) = 1$ and an outdegree of one $\delta^+(v) = 1$ [218].

viii. For a graph G, its **maximum degree** $\Delta(G)$ and its **minimum degree** $\delta(G)$ are the maximum and minimum degree of its vertices [221].

ix. A **cut vertex** is a vertex that would disconnect the graph when it is removed [219].

x. A **vertex cut** S (or, **vertex separator**, or **separating set**) is a **vertex subset** ($S \subset V$) for non-adjacent vertices (or subgraphs [224]) a and b, if the removal of the **vertex cut** S disconnects/separates the subgraphs "a and b into distinct connected components" [216].

(c) Notes about **edges**:

i. **Edges for directed graphs** are also called **arcs** [224].

- Alternate names for **edges of directed graphs** include **ordered pairs of vertices**, **arrows**, **directed edges**, **directed arcs**, and **directed lines**; see https://en.wikipedia.org/wiki/Directed_graph.
- A **directed edge** has a **head** and a **tail**; see https://en.wikipedia.org/wiki/Directed_graph. *What about* **directed hyperedges** *of* **directed hypergraphs***???*

ii. Alternate names for **edges of undirected graphs** include **unordered pairs of vertices**, and **lines**; see https://en.wikipedia.org/wiki/Directed_graph.

iii. An **endpoint** is a vertex connected by an edge [224].

iv. An **edge** has at least one endpoint (i.e., loop/self-loop). **Hyperedges** of **hypergraphs** can connect to more than 2 vertices; i.e., **hyperedges** can have more than 2 endpoints [224].

v. A **half-edge** is an edge with only one end, or **loose edge** has no ends [226].

vi. The **directed edge** $e = (u, v)$ [45, §B.4, pp. 1169] (of a directed graph) is **incident from**, or **leaves**, vertex u, and is **incident to**, or **enters**, vertex v.

vii. The **undirected edge** $e = (u, v)$ [45, §B.4, pp. 1169] (of an undirected graph) is **incident on** vertices u and v.

viii. **Adjacent vertices** are connected to each other by an edge [45, §B.4, pp. 1169]. For undirected graphs, this "**adjacency relation** is symmetric" [45, §B.4, pp. 1169]. Regarding directed graphs, this "**adjacency relation** is not necessarily symmetric" [45, §B.4, pp. 1169], since (u, v) may exist (hence, implying an adjacency relation) but (v, u) is nonexistent.

ix. References:

- [224]

(d) An **independent set** is a set of vertices S such that no pair vertices $(v_i, v_j), \forall i \forall j \in V, V = \{v_1, \ldots, v_n\}$ has an edge connecting them [219, 228].

i. That is, there does not exist any edge $\nexists e = (v_i, v_j), \forall i \forall j \in V, V = \{v_1, \ldots, v_n\}$.

(e) The **handshaking lemma** [221, 227] states that the number of vertices with odd degree in any undirected graph $G = (V, E)$ is even. $\sum_{v \in V} \deg(v) = 2|E|$.

2. A **subgraph** $S = (V_S, E_S)$ of a graph $G = (V_G, E_G)$ is another graph that includes a subset of the vertices and edges of G, such that $S \neq G, S \not\subset G, V_S \subset V_G, E_S \subset E_G$, and $(S \cap G) = S$ [224].
3. **Multi-dimensional networks**:
   (a) **Multi-dimensional networks** belong to type of **multi-layer networks** that have multiple types/kinds of relations [230].
   (b) A **multi-dimensional network** can be modeled with a **multipartite edge-labeled multigraph** [230, 232].

(c) An **unweighted multi-layer network** can be represented as a triple $G = (V, E, D)$, where (or, in which) V is a set of vertices, E is a dimension-specific set of edges connecting the vertices and each edge is represented by the triple (u, v, d) such that $u, v, \in V$ and $d \in D$ (or, $E = \{(u, v, d); u, v \in V, d \in D)$, and D is a set of dimensions or layers [230].

i. For an **unweighted, undirected multi-layer network**, the edges/links (u, v, d) and (v, u, d) are equivalent.

ii. For an **unweighted, directed multi-layer network**, the edges/links (u, v, d) and (v, u, d) are different/distinct.

iii. By convention, **unweighted multi-layer network** are not **multigraphs** in a given dimension; hence, "the number of [edges/]links between two nodes in a given dimension is either zero or one". "However, the total number of [edges/]links between two nodes across all dimensions is less than or equal to $|D|$."

(d) An edge of a **weighted multi-layer network** can be represented as a 4-tuple (or, quadruplet) $e = (u, v, d, w)$, "where w is the weight of the [edge/]link between [vertex] u and [vertex] v in the dimension d" [230]

i. **Weighted multi-layer networks**, like **unweighted multi-layer networks**, can also be represented as a triple $G = (V, E, D)$, where V is the set of vertices, E is the set of edges (each edge is represented by $e = (u, v, d, w)$), or $E = \{(u, v, d, w); u, v \in V, d \in D, w \in W\}$, W is a set of weights, and D is a set of dimensions or layers.

ii. A **weighted multi-layer networks**, where an edge is defined as $e = (u, v, d_1, \ldots, d_n, w)$, such that $d_1, \ldots, d_n \in D$, can model **multidimensional temporal networks**, or **multidimensional time-varying networks** [230].

(e) **Multidimensional network** [230]: https://en.wikipedia.org/wiki/Multidimensional_network.

(f) **Multipartite graph** [232]: https://en.wikipedia.org/wiki/Multipartite_graph

(g) **Temporal network**, or **time-varying network**: https://en.wikipedia.org/wiki/Temporal_network

(h) A **1-dimensional (1-D) network** has a **2-dimensional (2-D) adjacency matrix** with the size $V \times V$; A^i_j is an adjacency matrix that encodes edges/links/connections between vertices i and j.

(i) For a **multi-dimensional network** with $|D|$ dimensions, it has a **multi-layer adjacency tensor (or, 4-dimensional matrix, or 4-D matrix)** with the size $(V \times D) \times (V \times D)$; $M^{i\alpha}_{j\beta}$ is an **multi-layer adjacency tensor** that encodes edges/links/connections between vertices i in layer α and j in layer β [230].

4. **infinite graph**:
   (a) An **infinite graph** is a graph that is not finite. This can be worded better.

5. **extremal graphs**:
   (a) "**Extremal graph theory** studies **extremal (maximal or minimal) graphs** which satisfy a certain property. Extremality can be taken with respect to different graph invariants, such as order, size or girth."
   (b) https://en.wikipedia.org/wiki/Extremal_graph_theory

6. **random graphs**:
   (a) https://en.wikipedia.org/wiki/Random_graph
   (b) Wrongly referred to as probabilistic graph theory in https://en.wikipedia.org/wiki/Graph_theory

7. **Topological graph theory**:
   (a) https://en.wikipedia.org/wiki/Topological_graph_theory
8. **Geometric graph theory**:
   (a) https://en.wikipedia.org/wiki/Geometric_graph_theory

2.3.2 Graph Representations

Focus on **sparse graph representations**, which are common in modeling digital integrated circuits and neural networks (certain types), and **dense graphs** (e.g., neural networks).

For **sparse graphs**, use **adjacency list** (or **adjacency map** [86]) -based graph representations for **better memory efficiency** [225].

For **dense graphs** (i.e., $|E| \approx |V|^2$), use **adjacency matrix-based graph representation** for **faster access time** for finding edges at the expense of **worse memory efficiency** [225]; see https://en.wikipedia.org/wiki/Dense_graph. Also, see https://en.wikipedia.org/wiki/Dense_subgraph regarding dense subgraphs.

Hence, there exists a **trade-off between access time and member efficiency in graph representations**.

The ways to represent graphs are listed as follows:

1. **adjacency matrix**:
   (a)
   (b) References:
      i. [179, 225]
      ii. [85, §52.7, pp. 844; §52.6–§52.7, pp. 851–856]
      iii. Memory/Space efficient representation of dense graphs, especially when they are not multigraphs. It can represent dense undirected graphs most efficiently [85, §54, pp. 883].
      iv. For enumerating the edges outgoing from or incoming to a vertex in the graph, it takes $O(n)$ time, which slows down algorithms that need to enumerate edges outgoing from or incoming to each vertex in the graph [85, §54, pp. 883].
      v. [45, §22.1, pp. 589, 591–592] covers **adjacency matrix** graph representation. For dense graphs, the **adjacency matrix** graph representation enables quick determination of whether an edge exists between any pair of vertices in the graph [45, §22.1, pp. 589].
      vi. An adjacency matrix is a preferred representation for dense graphs, which have the property $|E| \approx |V|^2$. It is also preferred if there exists a need to frequently determine if there is an edge connecting any given pair of vertices (fast lookup, or fast access time) [45, §22.1, pp. 589]. The reference [45, §22.1, pp. 589–590] covers **adjacency matrix** graph representation.
      vii. [44, §5, pp. 78]
      viii. Its computational space complexity is $O(|V|^2)$ [87, §13.2, pp. 600] [86, §14.2, pp. 627] [225].
      ix. [87, §13.2.3, pp. 605–606] [86, §14.2, pp. 627; §14.2.4, pp. 633]
2. **adjacency list**:
   (a) [85, §52.7, pp. 850-854]
   (b) References:
      i. [225]

ii. [85, §52.7, pp. 854; §52.6–§52.7, pp. 851–856; §55, pp. 901–915]
iii. An adjacency list is a compact representation for sparse graphs [85, §54, pp. 883], which have the property $|E| \ll |V|^2$ [45, §22.1, pp. 589]. That is, it is "space efficient" [85, §55, pp. 901], or memory efficient.
iv. **It enables algorithms to efficiently (in regards to computational time complexity) enumerate edges outgoing from or incoming to each vertex in the graph faster than the adjacency matrix graph representation [85, §55, pp. 901]; the vertices in the graph are enumerated in a data-dependent order. The computational time complexity of enumerating the list of incoming edges and the list of outgoing edges can be reduced by using a set of incoming edges and a set of outgoing edges, due to the hashing properties of hash sets.**
v. [44, §5, pp. 79]
vi. Its computational space complexity is $O(|V| + |E|)$ [87, §13.2, pp. 600] [86, §14.2, pp. 627] [225]. While it is space efficient for sparse graphs, it is not space efficient for dense graphs.
vii. **[85, §55.1, pp. 904] mentions using a bucket mapping [85, §55.1, pp. 904; §50.3, pp. 829; §2.6.2, pp. 27; §50; §29.5, pp. 447] to implement the augmented adjacency sets that is used to store a set of incoming edges and a set of outgoing edges. The bucket mapping [85, §55.1, pp. 904; §50.3, pp. 829], or tagged bucket collection type [85, §2.6, pp. 25; §55.3.1, pp. 906], is based on a hash table. Bucket mapping, just like sets and mappings, are based on hash tables [85, §1.5, pp. 9].**
viii. [87, §13.2.2, pp. 603–604]
ix. [86, §14.2, pp. 627; §14.2.2, pp. 630–631]

3. **adjacency map**:
   (a) Its computational space complexity is $O(|V| + |E|)$ [86, §14.2, pp. 627].
   (b) [86, §14.2, pp. 627; §14.2.3, pp. 632; §14.2.5, pp. 634–637]

4. **edge list**:
   (a) Is this equivalent to the "**incidence list**" graph representation? Cite this!!!
   (b) [44, §5, pp. 78] describes the use of an unordered list to store the set of edges in the graph.
   (c) Its computational space complexity is $O(|V| + |E|)$ [87, §13.2, pp. 600] [86, §14.2, pp. 627].
   (d) [87, §13.2.1, pp. 600–602]
   (e) [86, §14.2, pp. 627; §14.2.1, pp. 628–629]

5. **incidence matrix**:
   (a) Its computational space complexity is $O(|V| \cdot |E|)$ [225].
   (b) References:
      i. [225]

Alternate graph representations that I am not exploring:
1. **distance matrix**: https://en.wikipedia.org/wiki/Distance_matrix

From [225], the adjacency list graph representation is more efficient in terms of computational space complexity than the adjacency matrix and incidence matrix graph representations.

Tables to compare the computational time complexities of graph methods between different graph representations and computational space complexities for data storage are found in: [85, §55.4 – §55.5, pp. 913].

References to cover:

1. [8, §4-§10, §19, §21-24, §25-§29]:
   (a) See [8, §3] regarding external sorting.
   (b) There exist a trade-off between computational time and space complexities in most data structures [8, §4.1.3, pp. 4-2], including graphs.
   (c) Use a "rooted tree with path compression" for "a simple and optimal" implementation of the union-find data structure [8, §4.1.4, pp. 4-3].
   (d) See [8, §7-§9] regarding graph algorithms.
   (e) See [8, §10] regarding external-memory algorithms and data structures.
   (f) See [8, §11] regarding average case analysis of algorithms, using probabilistic models and (probabilistic) techniques.
   (g) See [8, §12] regarding randomized algorithms.
   (h) See [8, §24] regarding complexity classes and measures for quantum computing.
   (i) See [8, §25] regarding parameterized algorithms.
   (j) See [8, §26] regarding machine learning.
   (k) See [8, §29] regarding distributed computing.
2. See [9]:
   (a) See [9, §25] regarding parallel algorithms.
   (b) See [9, §24] regarding "Algorithmic Techniques for Regular Networks of Processors".
   (c) See [9, §26] regarding self-stabilizing algorithms.
   (d) See [9, §28] regarding algorithms for implementing computing network protocols.
3. [85]:
   (a) Information regarding graph data structures and algorithms are found in [85, §51-§57].
   (b) See [85, §B.4] for information regarding expected time complexity.
      i. Use **expected time complexity** for *randomized algorithms*, or for *measuring computational time complexity based on random input*. For the latter case, we should specify the probability distribution for the possible inputs (e.g., uniform probability distribution).
   (c) [85, §52, pp. 843–844; §52.2, pp. 845–848] covers information on graph theory.
   (d) [85, §53.8] discusses the computational time complexity of the graph algorithms. **Update references in graph theory with this section!!!**
4. [45]:
   (a) [45, §B.4] covers graphs, and [45, §B.5] covers trees.
      i. Free trees are connected, undirected acyclic graphs [45, §B.5.1, pp. 173].
      ii. Rooted trees are acyclic graphs with a root [87, §13.1, pp. 598].
   (b) [45, §22.1] covers graph representations
   (c) [45, §D] covers basic matrix theory and matrix algebra.
   (d) [45, §21] and [86, §14.7] cover data structures for disjoint sets. **Update references in graph theory with this section!!!**
5. [44]:
   (a) [44, §5–§6] covers information on graph theory. **Update references in graph theory with this section!!!**
   (b)
6. [87]:
   (a) [87, §13.1] covers information on graph theory. **Update references in graph theory with this section!!!**
7. [86]:

   (a) [86, §14.1] covers information on graph theory. **Update references in graph theory with this section!!!**
   (b) [86, §14.7.3] covers union-find data structures.
8. [94]
9. [207]
10. almost linear time graph algorithms:
    (a) [27]
    (b) [38]???
11. [13]

Due to the lengthy coverage of graphs in [85, §52-§57], I would read the following references [44,45,87] first. Lastly, I would read [86] to implement the adjacency map representation of graphs.

2.3.3 Functions that need to be implemented

My ontology of graphs is based on whether they are directed, undirected, or mixed graphs [229].

Other ontologies of graphs are based on whether the graphs are:

1. weighted [85, 226]:
   (a) weighted graphs [85, §52, pp. 843; §51, pp. 842]
   (b) unweighted graphs [85, §52, pp. 843; §51, pp. 842]
2. density/sparsity of the edges in a graph:
   (a) Criteria for a graph to be considered as a dense graph:
      i. $m \geq \frac{n^2}{4}$, or comparable arbitrarily chosen cutoff [85, §52.2, pp. 848; §55, pp. 901].
   (b) Criteria for a graph to be considered as a sparse graph:
      i. [85, §52.2, pp. 848].

Required, or strongly preferred, data structures for implementing graphs:

1. Fibonacci heaps [45, §19, pp. 505–530], or "F-heaps" [44, §6, pp. 101]:
   (a) Used to implement the following (fast) graph algorithms [45, §V, pp. 481–482; §19, pp.505]:
      i. Dijkstra's single-source shortest path algorithm [85, §24.8, pp. 351; §57, pp. 925] [86, §14.6.2, pp. 667; §14.Chapter Notes, pp. 696] [87, §13.Chapter Notes, pp. 663].
      ii. all-pairs shortest paths algorithm [85, §24.8, pp. 351].
      iii. Prim's or Prim-Jarnik algorithm minimum spanning tree problem [85, §24.8, pp. 351; §57, pp. 925] [45, §23, pp. 624; §23-2, pp. 638] [86, §14.Chapter Notes, pp. 696] [87, §13.Chapter Notes, pp. 663].
   (b) "The Fibonacci heap is a forest, and not a tree" [85, §28.2, pp. 426].
   (c) When implementing priority queues, it uses constant time for merge operation instead of logarithmic time (leftist heap) or linear time (binary heap) [85, §24.4, pp. 346].
   (d) It has poor computational space complexity [85, §24.4, pp. 346–347].
   (e) See [85, §24.7, pp. 350] for the class hierarchy of priority queues and their implementations using binary heaps, leftist heaps, or pairing heaps (including Fibonacci heaps).
   (f) Instead of using the Fibonacci heap, use pairing heap to implement Prim's algorithm, or Prim-Jarnik algorithm, for minimum spanning trees [85, §24.8, pp. 351].
   (g) Certain types of relaxed heaps are faster than Fibonacci heaps for certain operations, especially for parallel programming [45, §19.Chapter notes, pp. 530].

i. Driscoll, Gabow, Shrairman, and Tarjan [96]. James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. Communications of the ACM, 31(11):1343–1354, 1988. [45, §29.Chapter notes, pp. 530; §Bibliography, pp. 1236] [86, §Bibliography, pp. 733] [87, §Bibliography, pp. 698].

(h) Also, see [86, §Bibliography, pp. 733] [87, §Bibliography, pp. 698]:
   i. M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," J. ACM, vol. 34, pp. 596–615, 1987.

(i) Analysis and treatment of Fibonacci heaps do not need understanding of binomial heaps as a prerequisite [45, §Preface, pp. xvi].

(j) See the following reference for the fastest implementation of Fibonacci heaps [45, §6.Chapter notes, pp. 169]:
   i. Rajeev Raman. Recent results on the single-source shortest paths problem. SIGACT News, 28(2):81–87, 1997. This is reference [291] in [45, §6.Chapter notes, pp. 169; §Bibliography, pp. 1246].
   ii. The bounds of EXTRACT-MIN and INSERT are $O(\min(\lg^{\frac{1}{4}+\epsilon} C, \lg^{\frac{1}{3}+\epsilon} n))$

2. circular array for topological sort (for DAGs) [85, §53, pp. 857].
3. queue (FIFO) for BFS [85, §53, pp. 857].
4. open addressing for strongly connected components (for directed graphs) [85, §53, pp. 857].
5. open addressing mapping for DFS [85, §53, pp. 857].

Note that for the following list of common functions to be implemented, functions that are implemented for the generic graph need not be specified/written again in the specifications and implementations of directed graphs and undirected graphs. They need to be specified/written again (only) if they need to be overridden.

Common functions that need to be implemented in (generic) graph data structures; they are instance methods, unless indicated otherwise (as static methods):

1. **Graph class**:
   (a) Standard constructor: *Graph(list_of_graph_vertices)*.
   (b) Standard constructor: *Graph(list_of_graph_vertices, list_of_graph_edges)* [85, §52.4, pp. 849; §52.5, pp. 850].
   (c) `is_adjacent(`v_u, v_v`)`: v_u and v_v are vertices (of the graph), if $v_u, v_v \in V_G$ [225] [87, §13.1.1, pp. 599].
   (d) `get_neighbors(`v_u`)`: v_u is a vertex (of the graph), if $v_u \in V_G$ [225] (for hypergraphs). Or, name it `opposite_vertex(`v_u`)` [87, §13.1.1, pp. 599], or `get_neighbor(`v_u`)` for graphs without hyperedges.
   (e) `add_vertex(`v_u`)`: v_u is a vertex (of the graph), and is added if $v_u \notin V_G$ [225] [86, §14.1.1, pp. 626] [87, §13.1.1, pp. 599] [85, §52.4, pp. 849; §52.5, pp. 850].
   (f) `remove_vertex(`v_u`)`: v_u is a vertex (of the graph), and is removed if $v_u \in V_G$ [225] [86, §14.1.1, pp. 626] [87, §13.1.1, pp. 599] [85, §52.4, pp. 849].
   (g) `has_vertex(`v_u`)` returns boolean `True` if $v_u \in V_G$. Else, return boolean `False` [85, §52.4, pp. 849; §52.5, pp. 850].
   (h) `add_edge(`$v_u, v_v, elem$`)`, or `add_edge(`$e_i, elem$`)` [85, §52.4, pp. 849; §52.5, pp. 850]: v_u and v_v are vertices (of the graph), and the edge is added if $v_u, v_v \in V_G$ and edge $e_i = (v_u, v_v) \notin E_G$ [225]. *elem* represents the attribute(s) of the edge object [86, §14.1.1, pp. 626] [87, §13.1.1, pp. 599].

(i) `remove_edge(`v_u, v_v`)`, `remove_edge(`e_i`)`, `remove_edge(`v_u, e_i`)` [85, §55.3.4, pp. 909]: v_u and v_v are vertices (of the graph), and the edge is removed if $v_u, v_v \in V_G$ and edge $e_i = (v_u, v_v) \in E_G$ [225] [86, §14.1.1, pp. 626] [87, §13.1.1, pp. 599] [85, §52.4, pp. 849].

(j) `has_edge(`v_u, v_v`)`, or `has_edge(`e_i`)`: v_u and v_v are vertices (of the graph), and the boolean `True` is returned if $v_u, v_v \in V_G$ and edge $e_i = (v_u, v_v) \in E_G$. Else, return boolean `False` [85, §52.4, pp. 849; §52.5, pp. 850].

(k) `get_edge(`v_u, v_v`)`: Returns the edge $e_i = (v_u, v_v) \in E_G$ from vertex v_u to vertex v_v, if the edge (v_u, v_v) exists; else, return `None` (or `Null`). For undirected graphs, there are no differences between edges (v_u, v_v) and edge (v_v, v_u), or between `get_edge(`v_u, v_v`)` and `get_edge(`v_v, v_u`)` [86, §14.1.1, pp. 626] [85, §52.4, pp. 849; §52.5, pp. 851]

(l) `get_vertex_attribute_x(`v_u`)`: return the value associated with vertex v_u's attribute x [225].

(m) `set_vertex_attribute_x(`$v_u, value$`)`: assigns/sets the value *value* associated with vertex v_u's attribute x [225].

(n) `get_edge_attribute_x((`v_u, v_v`))`, or `get_edge_attribute_x(`e_i`)`: return the value associated with the edge $e_i = (v_u, v_v)$'s attribute x [225].

(o) `set_edge_attribute_x((`v_u, v_v`), `$value$`)`, or `set_edge_attribute_x(`$e_i, value$`)`: assigns/sets the value *value* associated with the edge $e_i = (v_u, v_v)$'s attribute x [225].

(p) `get_all_vertices()` [87, §13.1.1, pp. 599]

(q) `get_all_edges()` [87, §13.1.1, pp. 599]:
   i. For directed graphs, provide a set/list of directed edges. Use the access functions for the head or tail to perform operations on the graph.

(r) `get_num_vertices()` returns the number of vertices in the graph $G = (V_G, E_G)$, or the cardinality of the set/list of vertices $|V_G|$ [85, §52.4, pp. 849; §52.5, pp. 851].

(s) `depth_first_search(string type)` returns a DFS ordering, which is one of the following DFS-based vertex orderings (preordering, postordering, or reverse postordering) [222]. *Implementation is found specifically for the abstract directed and undirected graph interfaces/headers/classes.*

(t) `breadth_first_search()` returns a BFS ordering [235]. *Implementation is found specifically for the abstract directed and undirected graph interfaces/headers/classes.*

(u) `is_updated()` returns boolean `True` if the graph has been updated since a graph enumeration algorithm has been executed [85, §53.1, pp. 858].

(v) `set_updated()` sets the `updated` flag to boolean `True` if the graph has been updated since a graph enumeration algorithm has been executed [85, §53.1, pp. 858].

(w) Optional methods:
   i. `boolean is_multigraph()` returns boolean `True` if the graph is a multigraph. Else, returns boolean `False` [85, §52.4, pp. 849].
   ii. `boolean set_multigraph(boolean mgraph)`, where the boolean flag `mgraph` indicates if the graph is a multigraph; if `mgraph` is true, the graph is a multigraph; else, the graph is not a multigraph.
   iii. `is_cyclic()` (or `has_cycles()`) returns boolean `True` if the graph G has cycles; else, it returns boolean `False` [85, §52.4, pp. 850].
   iv. `set_cyclic(boolean value)` set the `cyclic` attribute of the graph to boolean `True` if the graph G has cycles; else, it sets the `cyclic` attribute to boolean `False`. It returns nothing.
   v. `is_cycle(`E_{cycle}`)` returns boolean `True` if graph G contains the cycle E_{cycle}; else, it returns boolean `False` [85, §52.4, pp. 850].
   vi. `get_cycle()` [85, §52.4, pp. 850]: Not considered, since a graph can have multiple cycles.

vii. `boolean is_hypergraph()` returns boolean `True` if the graph is a hypergraph. Else, returns boolean `False`.

viii. `boolean set_hypergraph(boolean hgraph)`, where the boolean flag `hgraph` indicates if the graph is a hypergraph; if `hgraph` is true, the graph is a hypergraph; else, the graph is not a hypergraph.

ix. `boolean is_self_loop_pseudograph()` returns boolean `True` if the graph is a pseudograph (has self-loops). Else, returns boolean `False`.

x. `boolean set_self_loop_pseudograph(boolean self_loop)`, where the boolean flag `self_loop` indicates if the graph is a pseudograph (has self-loops); if `self_loop` is true, the graph is a pseudograph (has self-loops); else, the graph is not a pseudograph (has no self-loops).

xi. While directed and undirected graphs can implement these methods, their implementations are significantly different, and cannot be abstracted from both of these categories of graphs.

(x) Variables:

i. boolean `is_multigraph` to indicate if the graph is a multigraph [85, §53, pp. 857].

ii. boolean `hypergraph` to indicate if the graph is a hypergraph

iii. boolean `self_loop_pseudograph` to indicate if the graph allows self-loops to exist.

iv. set_of_cycles

v. boolean `has_been_modified` to indicate if the graph has been modified since a graph enumeration algorithm has been executed; boolean `true` indicates if it has been modified [85, §53.1, pp. 858].

vi. Ignore these variables:

A. DFS vertex ordering [85, §53, pp. 857], since there exists different types of DFS-based vertex ordering (preordering, postordering, or reverse postordering) [222]

2. **Vertex class**:
   (a) Accessor and mutator methods for properties of vertices $v_i, \forall i \in V_G = \{v_1, \ldots, v_m\}$
   (b)

3. **Edge class**:
   (a) Accessor and mutator methods for properties of edges $e_j, \forall j \in E_G = \{e_1, \ldots, e_n\}$
   (b)

Common functions that need to be implemented in **undirected graph** data structures; they are instance methods, unless indicated otherwise (as static methods):

1. **Graph class**:
   (a) `is_path(`e_{set}`)`, where e_{set} is a set of edges (or 2-tuples of vertices), and check if this ordered list/set of edges follows a sequence.
   (b) `are_adjacent_edges(`e_i, e_j`)` [87, §13.1.1, pp. 599]
   (c) `get_connected_components()` returns a set of connected components, where each connected component is defined by a set of vertices [85, §52.44, pp. 850].
   (d) `add_connected_components()` adds a connected component (defined by a set of vertices) to the set of connected components stored by the graph.
   (e) `get_number_of_connected_components()` returns the number of connected components in the graph G [85, §52.4, pp. 850].
   (f) `get_shortest_path_between_vertices(`v_u, v_v`)` returns the shortest path between vertex v_u and vertex v_v [85, §52.4, pp. 850].

(g) `get_set_of_shortest_paths_from_vertex(`v_u`)` returns the set of shortest paths between vertex v_u to each reachable vertex v_i (from v_u), such that a path from v_u to v_i exists, $\forall i \in$ set of paths from v_u [85, §52.4, pp. 850].

(h) `get_incident_edges(`v_u`)`:
   i. [87, §13.1.1, pp. 599]
   ii. [85, §52.5, pp. 851]

(i) `depth_first_search(string type)` returns a DFS ordering, which is one of the following DFS-based vertex orderings (preordering, postordering, or reverse postordering) [222].
   i. (May) require open addressing mapping for implementing DFS [85, §53, pp. 857].

(j) `breadth_first_search()` returns a BFS ordering [235].
   i. (May) require a queue (FIFO) for implementing BFS [85, §53, pp. 857].

(k) Optional methods:
   i. `boolean is_multigraph()` returns boolean `True` if the graph is a multigraph. Else, returns boolean `False` [85, §52.4, pp. 849].
   ii. `boolean set_multigraph(boolean mgraph)`, where the boolean flag `mgraph` indicates if the graph is a multigraph; if `mgraph` is true, the graph is a multigraph; else, the graph is not a multigraph.
   iii. `is_cyclic()` (or `has_cycles()`) returns boolean `True` if the graph G has cycles; else, it returns boolean `False` [85, §52.4, pp. 850].
   iv. `set_cyclic(boolean value)` set the `cyclic` attribute of the graph to boolean `True` if the graph G has cycles; else, it sets the `cyclic` attribute to boolean `False`. It returns nothing.
   v. `is_cycle(`E_{cycle}`)` returns boolean `True` if graph G contains the cycle E_{cycle}; else, it returns boolean `False` [85, §52.4, pp. 850].
   vi. `get_cycle()` [85, §52.4, pp. 850]: Not considered, since a graph can have multiple cycles.
   vii. `boolean is_hypergraph()` returns boolean `True` if the graph is a hypergraph. Else, returns boolean `False`.
   viii. `boolean set_hypergraph(boolean hgraph)`, where the boolean flag `hgraph` indicates if the graph is a hypergraph; if `hgraph` is true, the graph is a hypergraph; else, the graph is not a hypergraph.
   ix. `boolean is_self_loop_pseudograph()` returns boolean `True` if the graph is a pseudograph (has self-loops). Else, returns boolean `False`.
   x. `boolean set_self_loop_pseudograph(boolean self_loop)`, where the boolean flag `self_loop` indicates if the graph is a pseudograph (has self-loops); if `self_loop` is true, the graph is a pseudograph (has self-loops); else, the graph is not a pseudograph (has no self-loops).

(l) Variables:
   i. set_of_connected_components [85, §53, pp. 857].
   ii. set_of_cycles
   iii. Ignore these variables:
      A. DFS vertex ordering [85, §53, pp. 857], since there exists different types of DFS-based vertex ordering (preordering, postordering, or reverse postordering) [222]

2. **Vertex class**:
   (a) `is_incident_edge(`e_i`)` [87, §13.1.1, pp. 599].
   (b) `get_incident_edges()`:
      i. [86, §14.1.1, pp. 626].

ii. [87, §13.1.1, pp. 599]
iii. [85, §52.5, pp. 851]

(c) `is_adjacent_vertex(`v_v`)` [87, §13.1.1, pp. 599]
(d) `get_degree()` returns the number of edges incident to the current vertex [86, §14.1.1, pp. 626].
(e) `add_adjacent_vertex()`
(f) `add_adjacent_vertices()` for directed hypergraphs
(g) `remove_adjacent_vertex()`
(h) `remove_adjacent_vertices()` for directed hypergraphs
(i) Variables:
i. dict `adjacent_vertices` stores a dictionary of edges as key-value pairs, where the vertices are the keys for such pairs [86, §14.2, pp. 627; §14.2.3, pp. 632; §14.2.5, pp. 634–637].

3. **Edge class**:
(a) `get_endpoints()` [86, §14.1.1, pp. 626]. Or, name it `get_end_vertices()` [87, §13.1.1, pp. 599].
(b) `is_endpoint(`v_u`)` returns boolean `True` if vertex v_u is an endpoint of this edge. Else, return boolean `False`.
(c) `is_adjacent_edge(`e_i`)` [87, §13.1.1, pp. 599]. If the one of the endpoints of this edge is one of the endpoints on edge e, this edge is adjacent to edge e.
(d) `is_incident_on_vertex(`v_u`)` [87, §13.1.1, pp. 599].
(e) For two-endpoint edges, `is_opposite_vertex(`v_u`)` [87, §13.1.1, pp. 599].
(f) For two-endpoint edges, `get_opposite_vertex(`v_u`)` [87, §13.1.1, pp. 599] [86, §14.1.1, pp. 626].
(g) `is_incident_on(`v_u`)` [87, §13.1.1, pp. 599]. Is v_u the sink end-point of the edge [87, §13.1.1, pp. 599]?

Common functions that need to be implemented in **directed graph** data structures; they are instance methods, unless indicated otherwise (as static methods):

1. **Graph class**:
(a) `is_path(`e_{set}`)`, where e_{set} is a set of edges (or 2-tuples of vertices), and check if this ordered list/set of edges follows a sequence.
(b) `get_list_of_outgoing_edges(`v_u`)` returns a list of outgoing edges of vertex v_u [85, §52.4, pp. 849; §52.5, pp. 851].
(c) `get_list_of_incoming_edges(`v_u`)` returns a list of incoming edges of vertex v_u [85, §52.4, pp. 849; §52.5, pp. 851].
(d) `get_strongly_connected_components()` returns a set of strongly connected components, where each strongly connected component is defined by a set of vertices [85, §52.4, pp. 850].:

i. (May) require open addressing for implementing the algorithm/heuristic to search for strongly connected components [85, §53, pp. 857].

(e) `add_strongly_connected_components()` adds a strongly connected component (defined by a set of vertices) to the set of strongly connected components stored by the graph.
(f) `get_number_of_strongly_connected_components()` returns the number of strongly connected components in the graph G [85, §52.4, pp. 850].
(g) `get_topological_order()` returns an ordered set of vertices representing the graph G_{DAG} traversal using topological sort [85, §52.4, pp. 850]. If G_{DAG} is not a directed acyclic graph (DAG), throw a `GraphException`.

i. (May) require circular array to implement topological sort [85, §53, pp. 857].
ii. The algorithm for topological sort that is presented in [45, §22] is different [44, §5, pp. 89] from the algorithm for topological sort that is presented in [44, §5, pp. 75–78, 80] and in [120], which are more intuitive but not as simple [44, §5, pp. 89].

(h) `get_shortest_path_between_vertices(`v_u, v_v`)` returns the shortest path from vertex v_u to vertex v_v [85, §52.4, pp. 850].

(i) `get_set_of_shortest_paths_from_vertex(`v_u`)` returns the set of shortest paths from vertex v_u to each reachable vertex v_i (from v_u), such that a path from v_u to v_i exists, $\forall i \in$ set of paths from v_u [85, §52.4, pp. 850].

(j) `get_outgoing_edges(`v_u`)` [86, §14.1.1, pp. 626] [85, §52.5, pp. 851]

(k) `get_incoming_edges(`v_u`)` [86, §14.1.1, pp. 626] [85, §52.5, pp. 851]

(l) `depth_first_search(string type)` returns a DFS ordering, which is one of the following DFS-based vertex orderings (preordering, postordering, or reverse postordering) [222].
i. (May) require open addressing mapping for implementing DFS [85, §53, pp. 857].

(m) `breadth_first_search()` returns a BFS ordering [235].
i. (May) require a queue (FIFO) for implementing BFS [85, §53, pp. 857].

(n) Optional methods:
i. `boolean is_multigraph()` returns boolean `True` if the graph is a multigraph. Else, returns boolean `False` [85, §52.4, pp. 849].
ii. `boolean set_multigraph(boolean mgraph)`, where the boolean flag `mgraph` indicates if the graph is a multigraph; if `mgraph` is true, the graph is a multigraph; else, the graph is not a multigraph.
iii. `is_cyclic()` (or `has_cycles()`) returns boolean `True` if the graph G has cycles; else, it returns boolean `False` [85, §52.4, pp. 850].
iv. `set_cyclic(boolean value)` set the `cyclic` attribute of the graph to boolean `True` if the graph G has cycles; else, it sets the `cyclic` attribute to boolean `False`. It returns nothing.
v. `is_cycle(`E_{cycle}`)` returns boolean `True` if graph G contains the cycle E_{cycle}; else, it returns boolean `False` [85, §52.4, pp. 850].
vi. `get_cycle()` [85, §52.4, pp. 850]: Not considered, since a graph can have multiple cycles.
vii. `boolean is_hypergraph()` returns boolean `True` if the graph is a hypergraph. Else, returns boolean `False`.
viii. `boolean set_hypergraph(boolean hgraph)`, where the boolean flag `hgraph` indicates if the graph is a hypergraph; if `hgraph` is true, the graph is a hypergraph; else, the graph is not a hypergraph.
ix. `boolean is_self_loop_pseudograph()` returns boolean `True` if the graph is a pseudograph (has self-loops). Else, returns boolean `False`.
x. `boolean set_self_loop_pseudograph(boolean self_loop)`, where the boolean flag `self_loop` indicates if the graph is a pseudograph (has self-loops); if `self_loop` is true, the graph is a pseudograph (has self-loops); else, the graph is not a pseudograph (has no self-loops).

(o) Variables:
i. set_of_strongly_connected_components [85, §53, pp. 857].
ii. set_of_directed_cycles [85, §53, pp. 857].
iii. ordered set of vertices representing the graph G_{DAG} traversal using topological sort [85, §52.4, pp. 850; §53, pp. 857]. Can be omitted.

iv. Ignore these variables:
   A. DFS vertex ordering [85, §53, pp. 857], since there exists different types of DFS-based vertex ordering (preordering, postordering, or reverse postordering) [222]

2. **Vertex class**:
   (a) `get_outgoing_edges()` [85, §52.5, pp. 851] [86, §14.1.1, pp. 626]
   (b) `is_outgoing_edge(`e_i`)`
   (c) `get_out_degree()` returns the number of outgoing edges from the current vertex [86, §14.1.1, pp. 626].
   (d) `get_incoming_edges()` [85, §52.5, pp. 851] [86, §14.1.1, pp. 626]
   (e) `is_incoming_edge(`e_i`)`
   (f) `get_in_degree()` returns the number of incoming edges to the current vertex [86, §14.1.1, pp. 626].
   (g) `add_destn_vertex()`
   (h) `add_destn_vertices()` for directed hypergraphs
   (i) `add_src_vertex()`
   (j) `add_src_vertices()` for directed hypergraphs
   (k) `is_adjacent_to_destn_vertex(`v_v`)` [87, §13.1.1, pp. 599]. Does the edge $(v_{current}, v_v)$ exists?
   (l) `is_adjacent_from_src_vertex(`v_u`)` [87, §13.1.1, pp. 599]. Does the edge $(v_u, v_{current})$ exists?

3. **Edge class**:
   (a) `get_head_vertex()`, or `get_src_vertex()` [87, §13.1.1, pp. 599].
   (b) `get_head_vertices()`, or `get_src_vertices()`, for directed hypergraphs [87, §13.1.1, pp. 599].
   (c) `get_tail_vertex()`, `get_destn_vertex()` [87, §13.1.1, pp. 599].
   (d) `get_tail_vertices()`, or `get_destn_vertices()`, for directed hypergraphs [87, §13.1.1, pp. 599].
   (e) For two-endpoint edges, `is_opposite_vertex(`v_u`)` [87, §13.1.1, pp. 599]. *This is pretty useless.*
   (f) `is_adjacent_edge(`e_i`)` [87, §13.1.1, pp. 599]. That is, determine if the destination vertex of this edge is the source vertex for e_i; if they are, they are adjacent.
   (g) `is_incident_from_vertex(`v_u`)` [87, §13.1.1, pp. 599]. Is current edge incident from v_u? Basically, this is equivalent to `is_head_vertex(`v_u`)` (or, `is_src_vertex(`v_u`)`).
   (h) `is_incident_to_vertex(`v_v`)` [87, §13.1.1, pp. 599]. Is current vertex incident to v_v? Basically, this is equivalent to `is_tail_vertex(`v_v`)` (or, `is_destn_vertex(`v_v`)`).
   (i) `is_src_vertex(`v_u`)` returns boolean `True` if vertex v_u is a source vertex of this edge. Else, return boolean `False`.
   (j) `is_destn_vertex(`v_u`)` returns boolean `True` if vertex v_u is a destination vertex of this edge. Else, return boolean `False`.

Solvers for the following problems (or to perform the following functions) regarding:

1. **graph traversal**:
   (a) **breadth-first search**:
      i. https://en.wikipedia.org/wiki/Breadth-first_search
      ii. Also, see **BFS ordering**: https://en.wikipedia.org/wiki/Breadth-first_search

iii. Used for logic simulation, so that I can propagate values from one clock cycle to the next correctly. This assumes that the clock cycle is chosen as the minimum time period for the active/inactive pulse of the clock signal. See *Wikipedia*'s entry on duty cycle for vocabulary terms to describe the duty cycle of the clock signal as the ratio of the pulse width (or pulse active time) to the total period of the signal; reference https://en.wikipedia.org/wiki/Duty_cycle.
iv. [85, §53.4]
v. [45, §22.2]
vi. [87, §13.3–§13.4]
vii. [86, §14.3.3]

(b) **depth-first search**:
i. https://en.wikipedia.org/wiki/Depth-first_search
ii. **iterative deepening search**, or more specifically **iterative deepening depth-first search (IDS or IDDFS)**: https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search
iii. [85, §53.5]
iv. [45, §22.3]
v. [87, §13.3–§13.4]
vi. [86, §14.3.1–§14.3.2]

(c) **graph factorization**: ???

(d) References:
i. https://en.wikipedia.org/wiki/Graph_theory

2. **graph coloring**:
(a) **vertex coloring**
(b) **edge coloring**: https://en.wikipedia.org/wiki/Edge_coloring
(c) **four color theorem**, or the **four color map theorem**: https://en.wikipedia.org/wiki/Four_color_theorem
(d) References:
i. https://en.wikipedia.org/wiki/Graph_coloring
ii. [39]

3. **routing problems**:
(a) **shortest path problem**:
i. **Dijkstra's algorithm**: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
ii. **Bellman-Ford algorithm** (or, **Bellman-Ford-Moore algorithm**): https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
iii. **Ford-Fulkerson algorithm (FFA)**, or **Ford-Fulkerson method**: https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm
iv. **Categorize solutions** into those for **directed graphs** and **undirected graphs**. Also, determine solutions for common variants of the problem.
v. https://en.wikipedia.org/wiki/Shortest_path_problem
vi. [85, §55.2] references standard algorithms to find the shortest path.
vii. In addition, [85, §55.3] suggests using the in-tree data structure to represent the shortest path tree.
viii. [45, §24–§25]:
A. For the single-source shortest-paths problem, by implementing the min-priority queue with a Fibonacci heap, the running time (or computational time complexity) can be significantly improved [45, §24.3, pp. 662] [44, §6, pp. 101].
B. "In practice," other heaps (such as binary heaps) are used to implement the min-priority queue, which have better "constant factors hidden in the asymptotic notation" than Fibonacci heaps [44, §6, pp. 101].

C. For graphs with "weights [that] relatively small non-negative integers," there are faster "algorithms to solve the single-source shortest-paths problems" [45, §24.Chapter notes, pp. 682].
D. To solve the all-pairs shortest paths problem:
   - For each source vertex (or, vertex acting as the source) use Dijkstra's algorithm for the single-source shortest-paths problem along with an implementation of the min-priority queue with a Fibonacci heap [45, §25, pp. 684] [44, §6, pp. 101, 107]. This does not work well for dense graphs, due to "the constant factor induced by the Fibonacci heap" [44, §6, pp. 107].
   - For sparse graphs, Johnson's algorithm can solve the all-pairs shortest paths problem more efficient using an implementation of the min-priority queue with a Fibonacci heap (or, Fibonacci heap min-priority queue) [45, §25.3, pp. 700].
   - However, for dense graphs, "using d-ary min-heaps in shortest-paths algorithms on ϵ-dense graphs" can result in computational complexities or running times comparable to Fibonacci-heap-based algorithms [45, §25.Problems, pp. 706].

ix. [44, §6]
x. [87, §13.5]
xi. [86, §14.6]

(b) **longest path problem**:
   i. Note that the difficulty of the problem (in terms of **computational time complexity**) is different for different types of graphs:
      - E.g., for **undirected graphs**, it is **NP-hard**, while **linear time solutions** exist for **directed acyclic graphs (DAGs)**.
   ii. https://en.wikipedia.org/wiki/Longest_path_problem
   iii. [44, §5, pp. 80–85] covers longest path in a DAG.

(c) **minimum spanning tree**:
   -
   - **Prim-Jarník algorithm**:
     – Also, known as:
       * **Prim's algorithm**
       * **Jarník's algorithm**
       * **Prim-Dijkstra algorithm**
       * **DJP algorithm**
     – https://en.wikipedia.org/wiki/Prim%27s_algorithm
   - **Kruskal's algorithm**: https://en.wikipedia.org/wiki/Kruskal%27s_algorithm
   - https://en.wikipedia.org/wiki/Minimum_spanning_tree
   - [45, §23]:
     i. [45, §23.Chapter notes, pp. 642] addresses **spanning-tree verification**.
     ii. [45, §23.Chapter notes, pp. 642] also mentions faster algorithms for finding the minimum spanning tree (MST).
   - [87, §13.6]
   - [86, §14.7]

(d) **Steiner tree**:
   i. **rectilinear minimum Steiner tree (RMST)** problem: https://en.wikipedia.org/wiki/Rectilinear_Steiner_tree
   ii. https://en.wikipedia.org/wiki/Steiner_tree_problem

(e) **traveling salesperson problem** (NP-hard):
   i. **nearest neighbor algorithm**:
      - https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

- This is different from the k-nearest neighbors algorithm (k-NN); see https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm.

ii. https://en.wikipedia.org/wiki/Travelling_salesman_problem

(f) **strongly connected components (for directed graphs)**:
   i. https://en.wikipedia.org/wiki/Strongly_connected_component
   ii. See description in §2.3.4 regarding algorithms (and data structures) associated with directed graphs.
   iii. [45, §B.4, pp. 1170-1171]

(g) **connected components (for undirected graphs)**:
   i. https://en.wikipedia.org/wiki/Connected_component_(graph_theory)
   ii. [45, §B.4, pp. 1170]

4. **network flow**:
   (a) **max-flow min-cut theorem**: https://en.wikipedia.org/wiki/Max-flow_min-cut_theorem
   (b) **minimum-cost flow problem (MCFP)**: https://en.wikipedia.org/wiki/Minimum-cost_flow_problem
   (c) **maximum flow problems**: https://en.wikipedia.org/wiki/Maximum_flow_problem
   (d) **circulation problem**: https://en.wikipedia.org/wiki/Circulation_problem
   (e) References:
      i. **Flow network** (or **transportation network**): https://en.wikipedia.org/wiki/Flow_network
      ii.
      iii. *maximum flow*
         - [45, §26]

5. **graph partitioning**:
   (a) **force-directed graph partitioning**
   (b) **min-cut graph partitioning**
   (c) References:
      i. https://en.wikipedia.org/wiki/Connectivity_(graph_theory)

6. **graph-based floorplanning/placement**:
   (a) use **constraint graphs** for **graph-based floorplanning/placement**
   (b) References:
      i. https://en.wikipedia.org/wiki/Constraint_graph_(layout)

7. **covering problems**:

   (a) In graph theory, "**covering problems** are specific instances of **subgraph-finding problems**"; see https://en.wikipedia.org/wiki/Graph_theory

   (b) **Set cover problem**:
      i. https://en.wikipedia.org/wiki/Set_cover_problem
      ii. **hitting set problem**

   (c) **Vertex cover problem**:
      i. https://en.wikipedia.org/wiki/Vertex_cover
      ii. "A vertex cover is a set of vertices that includes at least one endpoint of each edge in the graph" [219]
      iii. "A vertex cover of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex [in] the set." Reference: https://en.wikipedia.org/wiki/Vertex_cover.

(d) **edge cover problem**:
   i. https://en.wikipedia.org/wiki/Edge_cover

(e) Related problems:
   i. **clique problem**: https://en.wikipedia.org/wiki/Clique_problem
   ii. **Covering/packing-problem pairs**, or **covering/packing dualities**: https://en.wikipedia.org/wiki/Linear_programming#Covering/packing_dualities
   iii. **Packing problems**:
      A. https://en.wikipedia.org/wiki/Packing_problems
      B. **Maximum set packing**: https://en.wikipedia.org/wiki/Set_packing
      C. **Maximum matching** (not graph matching), or **independent edge set**: https://en.wikipedia.org/wiki/Matching_(graph_theory)
      D. **independent set problem**, and **maximum independent set**: https://en.wikipedia.org/wiki/Independent_set_(graph_theory)
   iv. **Reconstruction conjecture**:
      - "Informally, the **reconstruction conjecture** in graph theory says that graphs are determined uniquely by their subgraphs."
      - https://en.wikipedia.org/wiki/Reconstruction_conjecture

(f) References:
   i. https://en.wikipedia.org/wiki/Covering_problems

8. graph matching:
   (a) https://en.wikipedia.org/wiki/Graph_matching

9. **graph decomposition problems**:
   (a) **arboricity**: https://en.wikipedia.org/wiki/Arboricity
   (b) **cycle double cover**: https://en.wikipedia.org/wiki/Cycle_double_cover
   (c) **graph factorization**: https://en.wikipedia.org/wiki/Graph_factorization
   (d) :
      i.
   (e) References:
      i. https://en.wikipedia.org/wiki/Graph_theory

10. **closure problem**:
   (a) https://en.wikipedia.org/wiki/Closure_problem

11. **spectral graph theory**:
   (a) "In mathematics, **spectral graph theory** is the study of the properties of a graph in relationship to the characteristic polynomial, eigenvalues, and eigenvectors of matrices associated with the graph, such as its adjacency matrix or Laplacian matrix."
   (b) References:
      i. https://en.wikipedia.org/wiki/Spectral_graph_theory
      ii. [196]
      iii. [32]
      iv. [54]
      v. [208]
      vi. [206]
      vii. [93]

12. **probabilistic graphical model** (**PGM**):

   (a) Also known as:
      i. **graphical model**
      ii. **structured probabilistic model**
   (b) References:
      i. https://en.wikipedia.org/wiki/Graphical_model
      ii. [16]
      iii. [26]
      iv. Cowell, Robert G.; Dawid, A. Philip; Lauritzen, Steffen L.; Spiegelhalter, David J. (1999). Probabilistic networks and expert systems. Berlin: Springer. A more advanced and statistically oriented book
      v. Jensen, Finn (1996). An introduction to Bayesian networks. Berlin: Springer.
      vi. Pearl, Judea (1988). Probabilistic Reasoning in Intelligent Systems (2nd revised ed.). San Mateo, CA: Morgan Kaufmann.

13. **quantum graph**:
   (a) "In mathematics and physics, a **quantum graph** is a linear, network-shaped structure of vertices connected by bonds (or edges) with a differential or pseudo-differential operator acting on functions defined on the bonds."
   (b) References:
      i. https://en.wikipedia.org/wiki/Quantum_graph
      ii. [130]

In summary, the graph data structures for generic graphs, undirected graphs, and directed graphs shall contain a compositional engine of graph problems (Class `compositional_engine`). That is, the classes for generic graphs, undirected graphs, and directed graphs shall only contain necessary attributes/properties/fields and methods, and methods implementing algorithms to solve graph problems are encapsulated by the `compositional_engine`.

Components of the `compositional_engine` are (listed in order of priority to implement):

1. `graph_traversal`
2. `routing_engine`
3. `visualization`:
   (a) VLSI floorplanning
   (b) VLSI placement
   (c) Visual representations of the graph
   (d) graph layouts
4. `graph_matching`
5. `graph_partitioning`
6. `graph_decomposition`
7. `graph_covering`
8. `graph_coloring`
9. `network_flow`
10. `probabilistic_graphical_models`
11. `network_science`
12. `graph_closure`
13. `spectral_graphs`

Solvers for the following problems (or to perform the following functions) regarding subgraphs, induced subgraphs, and minors:

1. **subgraph isomorphism problem**:
   (a) **Find a fixed graph as a subgraph in a given graph**:
      i. "graph properties are hereditary for subgraphs"... "A graph has a property if and only if all its subgraphs also have it"; see https://en.wikipedia.org/wiki/Graph_theory.
      ii. Finding a specific type/kind of maximal subgraph is an NP-complete problem, such as the largest complete subgraph.
   (b) Also, see **subgraph matching**.
   (c) **Graph isomorphism**: https://en.wikipedia.org/wiki/Graph_isomorphism
   (d) **Graph isomorphism problem**: https://en.wikipedia.org/wiki/Graph_isomorphism_problem
   (e) https://en.wikipedia.org/wiki/Subgraph_isomorphism_problem
2. **Finding induced subgraphs** in a given graph:
   (a) "graph properties are hereditary" for induced subgraphs... "A graph has a property if and only if all its induced subgraphs also have it"; see https://en.wikipedia.org/wiki/Graph_theory.
   (b) Finding a specific type/kind of **maximal induced subgraph** is an NP-complete problem:
      i. **Independent set problem**: Finding the largest **edgeless induced subgraph** (or **independent set**); see the following references:
         - https://en.wikipedia.org/wiki/Graph_theory
   (c) **Induced subgraph**: https://en.wikipedia.org/wiki/Induced_subgraph
3. **minor containment problem**:
   (a) Find a **fixed graph** as a minor of a given graph.
   (b) "A **minor** or **subcontraction of a graph** is any graph obtained by taking a **subgraph** and contracting some (or no) edges"... "A graph has a property if and only if all its **minors** [also] have it"
   (c) "[**Minor containment**] is related to graph properties such as planarity." See Wagner's Theorem about planar graphs.
   (d) References:
      i. https://en.wikipedia.org/wiki/Graph_theory
      ii. Graph minor: https://en.wikipedia.org/wiki/Graph_minor
4. **subdivision containment problems**:
   (a) "**Find a fixed graph as a subdivision of a given graph**":
      i. "A **subdivision** or **homeomorphism** of a graph is any graph obtained by subdividing some (or no) edges."
      ii. "**Subdivision containment** is related to graph properties such as planarity." See Kuratowski's Theorem and the Kelmans-Seymour conjecture about planar graphs.
   (b) References:
      i. https://en.wikipedia.org/wiki/Graph_theory
      ii. **Homeomorphism**: https://en.wikipedia.org/wiki/Homeomorphism_(graph_theory)#Subdivision_and_smoothing

Other common topics:

1. **network science**:
   - https://en.wikipedia.org/wiki/Network_science
   - **Interdependent networks**: https://en.wikipedia.org/wiki/Interdependent_networks

- **Modularity** (networks): https://en.wikipedia.org/wiki/Modularity_(networks)
- **Community structure**: https://en.wikipedia.org/wiki/Community_structure
- **Distance** (graph theory): https://en.wikipedia.org/wiki/Distance_(graph_theory)
- **Assortativity**: https://en.wikipedia.org/wiki/Assortativity
- **Degree distribution**: https://en.wikipedia.org/wiki/Degree_distribution
- **Centrality**: https://en.wikipedia.org/wiki/Centrality#Closeness_centrality
- **Betweenness centrality**: https://en.wikipedia.org/wiki/Betweenness_centrality

Other problems and types of graphs, not of comparable importance:
1. nearest neighbor graph (NNG): https://en.wikipedia.org/wiki/Nearest_neighbor_graph

The *GraphException* class is a child class of the Exception class. See https://github.com/eda-ricercatore/gulyas-scripts/blob/master/notes/python.md#exception-handling for information about implementing the *GraphException* class [85, §52.4, pp. 849].

2.3.3.1 Decision Decisions in Implementing Graphs

Design decisions in implementing graphs:
1. Assume that edges of the graphs have no identities [231].:
   (a) This makes defining the constructor and instantiating instances of the graph data structures easier. Since there is no need to define a function (for each graph class) that assigns each edge to:
      i. An unordered pair of vertices or (for endpoint nodes) for undirected graphs.
      ii. Its designation and target nodes for directed graphs.
   (b) The smaller the tuple for defining the constructor of the graph class, the easier it is to maintain the graph module/package, extend it, and use it.

2.3.4 Directed Graphs, and Directed Acyclic Graphs

Notes on directed graphs (digraphs), and directed acyclic graphs (DAGs):
1. Functions to implement, and solves to solve the following problems:
   (a) transitive closure:
      i. [87, §13.4.2] [86, §14.4]
   (b) For DAGs:
      i. **topological sort**:
         - Also known as: **topological sorting**, or **topological ordering**
         - https://en.wikipedia.org/wiki/Topological_sorting
         - [85, §53.6]
         - [45, §22.4]
         - [44, §5, pp. 75–80]
         - [86, §14.5.1]
      ii. **strongly connected components**:
         - [45, §22.5]
      iii. **precedence graphs**, **conflict graphs**, or **serializability graphs**
         - "Used in the context of concurrency control in databases"; see https://en.wikipedia.org/wiki/Precedence_graph
         - [85, §53.6]
         - [85, §52.1, pp 844]
      iv. algorithms, heuristics, and meta-heuristics to optimize:
         - BDDs; see §2.3.4.1.
         - AIGs; see §2.3.4.2
         - MIGs; see §2.3.4.3

2.3.4.1 Binary Decision Diagrams (BDDs)

Resources for BDDs:
1. https://en.wikipedia.org/wiki/Binary_decision_diagram

2.3.4.2 AND-Inverter Graphs (AIGs)

Resources for AIGs:
1. https://en.wikipedia.org/wiki/And-inverter_graph

2.3.4.3 Majority-Inverter Graphs (MIGs)

Resources for MIGs:
1.

2.3.5 Undirected Graphs

Notes about undirected graphs are available in the subsection on graph theory, §2.3.1.

2.3.6 Resources for Graphs

Resources for graphs:
1. "The Stanford GraphBase: A Platform for Combinatorial Computing":
   (a) [119]
   (b) https://www-cs-staff.stanford.edu/~knuth/sgb.html
2. **Graph analysis** using techniques from **linear algebra**, or **linear algebra in graph theory**:
   (a) See references [14, 15, 25, 32, 100]. Also, see [131].
   (b) E.g., "the vertex space of a graph is [represented by] the vector space" that has a set of basis vectors that correspond to the vertices of the graph [219]
   (c) For edge space and vector space, see https://en.wikipedia.org/wiki/Edge_space.
3. *Wikipedia*:
   (a) https://en.wikipedia.org/wiki/List_of_algorithms#Graph_algorithms
   (b) **Gallery of named graphs**: https://en.wikipedia.org/wiki/Gallery_of_named_graphs
   (c) **List of graph theory topics**: https://en.wikipedia.org/wiki/List_of_graph_theory_topics
   (d) **Glossary of graph theory terms**: https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms
   (e) :
   (f) :
   (g) :
   (h) :
   (i) :
   (j) :
   (k) :
   (l) :
   (m) :
   (n) **Graph algebra**: https://en.wikipedia.org/wiki/Graph_algebra
   (o) **Algebraic graph theory**:
      i. https://en.wikipedia.org/wiki/Algebraic_graph_theory
      ii. Can use linear algebra (includes spectral graph theory), group theory, and the study of graph invariants.

2.4 Notes on Data Structures for External Memory Computation

See [87, §14.3] and [86, §15.3] for data structures that can be used for computing with external memory. [87, §14.2.1] and [86, §15.2.1] defines what is **internal memory** (i.e., **main memory**, or **core memory**) and **external memory** (e.g., **hard disks**/drives, CD drives, DVD drives, and tapes).

Chapter 3

Optimization

3.1 Benchmarks for Optimization

A collection of "optimization solvers" and benchmarks are available at [69].

Benchmarks for optimization problems:

1. MIPLIB 2010 – Mixed Integer Programming Library version 5 [121]. See [2] for publications associated with this set of benchmarks (or benchmark set).
2.

3.2 Notes on Using Optimization Tools

Optimization problems in EDA can be solved via optimization engines that I implement or external (i.e., third-party) optimization solvers.

Regarding external optimization solvers, some of them use *Algebraic Modeling Languages (AML)* [214] to model the optimization problem computationally. These optimization solvers can solve optimization problems that are formulated as computational models in a specific AML representation.

I am avoiding the use of external optimization solvers that require paid licenses. Hence, any external optimization solvers that I would use are either open-source software (or rather, free/libre/open-source software, FLOSS) or software that have free academic licenses.

Solvers that use an AML, or several AMLs, in their software interface are:

1.

For a list of optimization solvers/tools, see §3.5.

3.3 Robust Linear Programming

During the "lab meeting" on Friday, December 4, 2015, Prof. Jiang Hu told me that I can transform a robust linear programming into a standard/"standard" linear programming problem. He told me to look at [24] and its references.

3.4 Discrete Optimization

Discrete optimization is classified into the following categories [96, 127, 215]:

1. combinatorial optimization
2. integer programming

3.5 Optimization Solvers

A (brief) description of optimization solvers (not restricted to solvers for mathematical programming), including linear programming solvers, is provided as follows in §3.5.1 and §3.5.2.

3.5.1 Accessible Optimization Solvers

External optimization solvers that are open-source software or provide free academic licenses:

1. *LocalSolver* [105]:
   (a) Hybrid solver for optimization problems
   (b) Properties of the solver [105, Product: Overview]:
      i. "next-generation, hybrid mathematical programming solver"
      ii. solve "ultra-large real-life nonlinear problems"
      iii. solve problems in a "model-and-run fashion without any tuning"
      iv. reliable and robust solver: **Define reliability and robustness for solvers of optimization problems.**
      v. dynamically combines solutions from various optimization approaches and resolves them via a hybrid neighborhood search approach
      vi. solver engines:
         A. "local search techniques"
         B. "constraint propagation techniques"
         C. "inference techniques"
         D. linear programming solver/techniques
         E. mixed-integer programming solver/techniques, including mixed-integer linear programming (MILP) solver/techniques
         F. nonlinear programming solver/techniques
         G. combined pure and direct local search techniques
      vii. is based on the *LocalSolver Programming language* (LSP) for mathematical modeling
      viii. has lightweight object-oriented APIs
   (c) From [105, Support Center: Example tour], *LocalSolver* can solve continuous and discrete/combinatoria optimization problems:
      i. continuous optimization problems:
         A. minimization of the Branin function: find the minimal point of the Branin function, within a specified domain
         B. optimal bucket design: minimization of a bucket encapsulating/covering the rod/cylinder
         C. Steel mill slab design: mathematical programming
      ii. discrete optimization problems:
         A. car sequencing: scheduling problem, or assignment problem.
         B. Flowshop: scheduling problem
         C. knapsack problem
         D. max-cut problem
         E. Quadratic Assignment Problem (QAP)

F. Steel mill slab design: integer programming
G. Travelling salesman problem
H. Vehicule routing problem

(d) Its technical documentation can be found at: http://www.localsolver.com/documentation/index.html [104].

2. Stanford University:
   (a) Systems Optimization Laboratory researchers, "SOL Optimization Software," from *Stanford University: School of Engineering: Department of Management Science and Engineering: Systems Optimization Laboratory*, Stanford, CA, 2015. Available online at: http://web.stanford.edu/group/SOL/download.html; last accessed on December 14, 2015.
      i. "Iterative solvers for sparse $Ax = b$: SYMMLQ, MINRES, MINRES-QLP, cgLanczos, CRAIG"
      ii. "Iterative solvers for sparse least-squares problems: LSQR, LSMR, CGLS, covLSQR, LSRN"
      iii. "Sparse and dense LU factorization (direct methods): LUSOL, LUMOD"
      iv. "Sparse optimization: ASP"
      v. "Optimization with convex objective and linear constraints: PDCO (including sparse optimization)"
      vi. "Convex optimization in composite form: PNOPT"
      vii. "Fortran 90 quad-precision dotproduct of double-precision vectors: qdotdd"

3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.

Additional notes:

1. For mixed-integer programming, check performance comparisons on MIPLIB benchmarks (http://www.localsolver.com/news.html?id=32)

3.5.2 Not Accessible Optimization Solvers

External optimization solvers that require paid licenses:

1. Stanford University:
   (a) Systems Optimization Laboratory researchers, "SOL Optimization Software," from *Stanford University: School of Engineering: Department of Management Science and Engineering: Systems Optimization Laboratory*, Stanford, CA, 2015. Available online at: http://web.stanford.edu/group/SOL/download.html; last accessed on December 14, 2015.

i. LSSOL
ii. MINOS
iii. NPSOL
iv. QPOPT
v. SNOPT
vi. SQOPT

2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.

Chapter 4

Mathematics

Math symbols that I use frequently:

1. $\mathbb{N}$
2. $\sum_{i=1}^{n}$
3. $f(x) = \lim_{n\to\infty} \frac{f(x)}{g(x)}$
4. $\varnothing$
5. q

A 3×3 matrix: $\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{pmatrix}$

Here is an equation:

$$\iint_{\Sigma} \nabla \times \mathbf{F} \cdot \mathrm{d}\mathbf{\Sigma} = \oint_{\partial\Sigma} \mathbf{F} \cdot \mathrm{d}\mathbf{r}. \tag{4.1}$$

Here is an equation that is not numbered.

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

Here is the set of Maxwell's equations that is numbered.

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0} \tag{4.2}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{4.3}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{4.4}$$

$$\nabla \times \mathbf{B} = \mu_0 \left(\mathbf{J} + \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t} \right) \tag{4.5}$$

$$\begin{aligned} &\text{minimize} \sum_{\mathrm{i}=1}^{\mathrm{c}} \mathrm{c_i} \cdot \mathrm{x_i} \\ &\underline{x} \in S \\ &\text{subject to :} \\ &x_1 + x_4 = 0 \\ &x_3 + 7 \cdot x_4 + 2 \cdot x_9 = 0 \end{aligned}$$

$$f(n) = \begin{cases} case - 1 & : \text{n is odd} \\ case - 2 & : \text{n is even} \end{cases} \tag{4.6}$$

Proof. This is a proof for BLAH ... □

Theorem 4.1. *TITLE of theorem. My theorem is...*

Axiom 4.1. *TITLE of axiom. Blah...*

Cases of putting a bracket/parenthesis on the right side of the equation.

$$\left.\begin{aligned} B' &= -\partial \times E, \\ E' &= \partial \times B - 4\pi j, \end{aligned}\right\} \quad \text{Maxwell's equations}$$

Labeling an arrow: $\xrightarrow{ewq}$

Chapter 5

Statistics

Chapter 6

C++ Resources

Quick advice: `Learn how to use` *C++1y* `features, including those of C++11, C++14, and C++17.` Older *C++* versions include *C++98* and *C++03*.

Reference: Free Software Foundation contributors, "C++1y/C++14 Support in GCC," from *GCC, the GNU Compiler Collection: GCC Projects*, Free Software Foundation, Boston, MA, November 14, 2015. Available online at: https://gcc.gnu.org/projects/cxx1y.html; last accessed on January 25, 2016.

Add references for this chapter!

Books/references that cover the following modern *C++* standards:

1. *C++11*:
   - [213]
   - [200]
   - [201]
   - [31]
   - [36]
   - [124]
   - [128]
   - [129]
   - [137]
   - [136]
   - [155]
   - [158]
   - [182]
   - [191]
   - [199]
   - [239]
   - [106]
   - [5]
   - [30]
   - [60]
   - [82]

- [89]
- [92]
- [101]
- [113]
- [142]
- [167]
- [181]
- [211]
- [20]
- [91]

2. *C++14*:
   - [88]
   - [95]
   - [107]
   - [141]
   - [148]
   - [66]
   - [138]
   - [139]
   - [146] – Important.
   - [204]
   - [56]
   - [61]
   - [79]
   - [90]
   - [102]
   - [203]
   - [202]
   - [72]
3. *C++17*:
   - [140]
   - [62]
   - [108]
   - [149]
   - [169]
   - [175]
   - [194]
4. Other books/references that may over modern *C++*:
   - [17]
   - [154]
   - [58]
   - [160]
   - [197]
   - [212]
   - [18]

- [55]
- [71]
- [168]
- [70]
- [145]
- [162]
- [192]
- [236]
- [57]
- [65]
- [76]
- [83]
- [133]
- [134]
- [135]
- [157]

5. Other helpful resources for *C++*:
   - [75] used with *R*
   - [123]
   - [147]

6.1 Resources for C++ and Notes About C++

Some *C++* resources are:

1. Online *C++* tutorial reference: [46, 47, 49, 195]: http://www.cplusplus.com/reference/stl/
2. Pointers to functions [47]: http://www.cplusplus.com/doc/tutorial/pointers/
3. An online *C++* reference: [50].
4. From an online *Marshall Cline* resource: [40–42]
5. Books on *C++*: [122, 165]; processed for *C++* templates
6. Embedded *C++*: [116]
7. GUI and other graphics with *C++*: [165]
8. Summaries of *C++*: [112]

Some *C++* STL resources are:

1. Online *C++* STL reference: [48]: http://www.cplusplus.com/reference/stl/
2. [50]: http://en.cppreference.com/w/cpp/container
3. From an online *C++* STL resource from SGI (formerly, Silicon Graphics, Inc.): [98, 99].
4. Another online C++ STL reference: [150]: http://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm
5. Other online *C++* STL references: [171, 172].
6. http://www.cs.wustl.edu/~schmidt/PDF/stl4.pdf

Books to classify:

1. C++ programming: [101, 166, 174, 180, 186, 187]

References for *C++* libraries of interest:

1. *Boost C++*: [114, 153, 163, 183]

2. *Other libraries that can be considered include: cpp-netlib: The C++ Network Library; nana; FLTK; gtkmm; Qt; evince; glibmm; cairomm; opencv; Ogre3D; OpenGL; CGAL; QuantLib; Google Test; and cppunit.*
3. E.g., see http://en.cppreference.com/w/cpp/links/libs [50, Useful resources: List of C++ libraries – A list of open source C++ libraries].

C++ topics:

1. Function objects:
   (a) https://en.wikipedia.org/wiki/Functional_(C%2B%2B)
   (b) http://stackoverflow.com/questions/356950/c-functors-and-their-uses
   (c) http://www.cprogramming.com/tutorial/functors-function-objects-in-c++.html
   (d) [113, pp. 233–243]
   (e) [166, pp. 327–332, 885, 922–931, 947]
   (f) [186, pp. 126–129]. Function pointers are pointers to functions; note that these functions are not variables.
2. Strings:
   (a) [200], Chp 23
   (b) [198], Chp 23
   (c) [90], Chp 18
   (d) [5], Chp 19
   (e) [101, pp. 56–60, string data types, variable vs. literal; strings and string class, 13, 82–87, 320–324, 363–365, 496]
   (f) [113, 655–716]
   (g) [180, pp. 64–67, 320–325, 465–482]
   (h) [184, §14.2]
   (i) [166, pp. 114–131]
   (j) [74], Chp 1
   (k) [97]:
      i. The `put` pointer points "to the next free byte in the `stringstream`." That is, it "holds the address of the next byte in the output area of the" `stringstream`. When the `stringstream` is empty, the `put` pointer points to the beginning of the `stringstream` buffer. [97, §9.8].
      ii. "The type of the `put` pointer" does not matter to the software developer(s), since they "cannot access it directly" [97, §9.8].
      iii. "The `get` pointer holds the address of the next byte in the input area of the stream, or the next byte we get if we use >> to read data from the `stringstream`" [97, §9.9].
      iv. "The `end` pointer indicates the end of the `stringstream`. Attempting to read anything at or after this position will cause the read to fail because there is nothing else to read" [97, §9.9].
      v. Developers only have to know about how `put`, `get`, and `end` pointers work. They do not have to know the actual representation of these pointers [97, §9.9].
      vi. The `stringstream` object acts as a buffer, and is "an area of allocated memory" ("by the stringstream member functions") [97, §9.9].
   (l) [143]:
      i. C strings (or C-strings, or C-style strings) are null-terminated strings (arrays of characters that each end with a terminating "null character" with ASCII value 0) and are arrays of characters; the "null character" is usually represented by the literal character '\0'. "However, an array of `char` is NOT by itself a C string."

ii. "Since `char` is a built-in data type, no header file is required to create a C string. The C library header file <`cstring`> contains a number of utility functions that operate on C strings."
iii. "It is also possible to declare a C string as a pointer to a `char`: char* s3 = "hello"; " It creates a character array with just enough memory space (in the heap) to store the null-terminated string. The address of the string's first character is placed in the `char` pointer *s3*. When this improperly used, it can corrupt program memory or cause run-time errors.
iv. "[Use] the `C` library function `strlen()`" to determine "the length of a `C` string." It returns an unsigned integer representing the number of characters in the string, excluding the terminating null character.
v. Relational operators (such as ==, ! =, >, <, >=, <=) compare the addresses of the first characters in the two string operands (as the array names are treated as pointers), instead of the contents of these strings.
vi. "Use the `C` library function `strcmp()`" "to compare the contents of two `C` strings." The input arguments of this function are two pointers to `C` strings.
vii. "Use the `C` library function `strcpy()`" to assign a string to a C string or change its contents. The `strcpy()` function accepts a pointer to the C string as the first input argument, and a pointer to the contents of a valid C string or string literal (i.e., a character) as the second input argument. The `C` library function `strcat()` has the same input arguments as `strcpy()`, and is used for concatenating two strings.
viii. C strings can be used as input parameters or the return type. They are specified as `char[]` or `char`*.
ix. "A `C++` string is an object of the class string, which is defined in the header file <string> and which is in the standard namespace." The variable name of a `C++` string is a pointer to the first character of the string; the variable name contains the address of the string's first character. The `C++` string is a dynamically-allocated array of characters.
x. "[Use] the string class methods `length()` or `size()`" to determine "the length of the `C++` strings."
xi. To improve memory efficiency and reduce memory usage, explicitly *pass a string object*. Else, the `C++` string objects are pass and returned by value, which involves making a copy of the string object.
xii. Concatenate C++ strings, C strings, and string literals in any order using the "+" operator.
xiii. Convert a C++ string into a C string via the `c_str()` function of the `string` class. The `c_str()` function returns a pointer to the array of characters representing the string. If the C++ string is not null-terminated, a null character is appended to the new C string. The returned C string "can be used, printed, copied, etc." but not be modified.
xiv. Since programming with arrays can enbug the code more easily, the use of C++ string is (strongly) recommended for use. This is because the properties of a2
xv. When a C string is required by a function, convert the C++ string into a C string (as aforementioned). Instances in which a C string have to be converted into a C++ string are:
    A. Strings passed into `main()` as C strings from the command line argument.
    B. Functions for file input/output operations require filenames to be specified as C strings.
    C. The C++ string class does not have the equivalent functions of certain C string library functions.
    D. Unlike C++ strings, C strings can be serialized in binary format without requiring a bunch of extra code to be written.
xvi. The function `atoi` converts a string to an integer. Similar functions for converting

strings into numbers are: `atol` and `atof`. The C++ STL does not have a `itoa` function to convert a number to an integer. However, some compilers supports this function in the *C Standard General Utilities Library*.

(m) The function `strtol` converts a string into a long integer:
   i. See http://www.cplusplus.com/reference/cstdlib/strtol/.
   ii. [48, <cstdlib> (stdlib.h) – C Standard General Utilities Library: `strtol` function]

(n) Danny Kalev, "String Streams," in *InformIT: The Trusted Technology Learning Source: Articles: Programming: C/C++ Articles: InformIT C++ Reference Guide*, Pearson Education, Indianapolis, IN, January 1, 2003. Available online at: http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=72; last accessed on November 13, 2015.
   i. Static buffers (via *atoi()*, *sprintf()*, or *sscanf()* from the *<stdio.h>*) for type conversions can cause buffer overflow and do not provide adequate type safety (i.e., adequate type checking mechanism). This can be mitigated via *stringstreams*.

3. IO Streams:
   (a) [90], Chp 12
   (b) [200], Chp 10-11
   (c) [128], Chp 8
   (d) [5], Chp 28
   (e) [82, chp. 12].
   (f) [101, Chp. 8, pp. 351–388; pp. 4, 12–15, 49–51, 150–154, 352, 361–365, 497–498]
   (g) [113, pp. 743–847]
   (h) [167], Chp 17q
   (i) [83, Chp. 13].
   (j) [81], Chp 12.
   (k) [180, §2.2; Chp. 6]
   (l) [198], Chp 10-11
   (m) [184, §14.3-14.8]
   (n) [166, pp. 262–273]
   (o) [74], Chp 2
   (p) [156], Chp 16
   (q) [188], Chp 21
   (r) [187, Chp. 18, pp. 417–450; Appendix A, pp. 563–580]
   (s) [210], Chp 10
   (t) [186, Chp. 8–9, pp. 187–235; Chp. 20–21, pp. 511–568]

4. Templates:
   (a) [90], Chp 11,21
   (b) [200], Chp 19
   (c) [128], Chp 16
   (d) [5], Chp 29
   (e) [82, §16.2–§16.4]
   (f) [113, pp. 13, 26–27, 33–34, 36, 62, 68, 1024]
   (g) [83, §16.2–§16.4]
   (h) [180, Chp. 17]
   (i) [198], Chp 19
   (j) [165, §6.16, pp. 193–195]
   (k) [1], book

(l) [166, pp. –]
(m) [74], Chp 3
(n) [156], Chp 24
(o) [188], Chp 18
(p) [187, Chp. 16, pp. 375–394]
(q) [209], book
(r) [4], book; typelist - Chp 3
(s) [210], Chp 6
(t) [73], Chp 16
(u) [186, Chp. 18, pp. 461–487]
(v) References:
i. [152] recommends including the implementation file (`.cpp`), instead of the header file (`.hpp`) in any *C++* file that uses the *C++* template.
ii. [21] argues for separating the *C++* template definitions in the header file (`.hpp`) from the *C++* template implementations in the implementation file (`.cpp`).
iii. "Although standard C++ has no such requirement, some compilers require that all function and class templates need to be made available in every translation unit they are used. In effect, for those compilers, the bodies of template functions must be made available in a header file. To repeat: that means those compilers won't allow them to be defined in non-header files such as .cpp files.... There is an export keyword which is supposed to mitigate this problem, but it's nowhere close to being portable." [84]
iv. [22] argues that you can merge the definition and implementation in a single file, which ends with the `.hpp` file extension.
v. [63,67] argue that both the definition and the implementation of *C++* templates should be placed together in the *C++* header file, with white space
vi. This is a list of different implementations of a C++ template:
A. Implemented with *C++* template definition and implementation in one file, where the definition and implementation are clearly separated/distinguished/demarcated. The implementation is appended to the end of the definition, with some white space to separate them [63, 67, 84].
B. Implemented with merged *C++* template definition and implementation in one file [22].
C. Implemented with *C++* template definition and implementation separately in different files: a *C++* header file and a *C++* implementation file. Any file that uses this *C++* template will have to import the *C++* implementation file, instead of the *C++* header file [21, 152]. `I shall use this method. This clearly decouples the C++ template definition and implementation separately into different files, and keeps things modular in my software architecture. It prevents templates from being huge monoliths.`

5. Debugging:
(a) [74], Chp 11 (especially memory management problems, pp. 533)

6. STL, books on *C++* STL:
(a) [82, §16.5, 983–996]
(b) [113]
(c) [180, Chp. 18, pp. 943–998]
(d) [170]
(e) [166, Chp. 16, pp. 877–922, 930–940]

(f) [187, Chp. 21, 499–545]
(g) [173]
(h) [111]

7. STL containers:
   (a) [90], Chp 15-16
   (b) [128], Chp 9,11
   (c) [5], Chp 18
   (d) [167], Chp 16
   (e) [77]:
      i. `vector<int> v(10);` *// Create an int vector of size 10.*
      ii. `v[5] = 10;` *// Target of this assignment is the return value of operator[].*
      iii. Determine how to operate with a pointer to a vector of pointers:
         A. vector<object* > *connections;
         B. object* oneObject = new object;
         C. connections–>push_back(oneObject);
         D. (*connections)[0]–>Initialize(index);
   (f) [180, §18.2, pp. 960–977]
   (g) [170], book
   (h) [184, Chp. 8]
   (i) [189], Chp 8
   (j) [74], Chp 4
   (k) [156], Chp 25
   (l) [210], Chp 7
   (m) [186, Chp. 24, pp. 625–691; Chp. 25–38, pp. 695–927]

8. STL algorithms:
   (a) [90], Chp 15,17
   (b) [128], Chp 10
   (c) [5], Chp 18
   (d) [167], Chp 16
   (e) [180, §18.3, pp. 977-991]
   (f) [170], book
   (g) [74], Chp 5
   (h) [156], Chp 25
   (i) [210], Chp 7

9. Function addresses:
   (a) [200], Chp 8
   (b) [198], Chp 8
   (c) [166, pp. 330–331]
   (d) [73], Chp 3, pp. 213

10. Dynamic memory management problems:
   (a) [90], Chp 10,22
   (b) [128], Chp 12,13
   (c) [5], Chp 14
   (d) [82, §13.9, 750–754].
   (e) [167], Chp 9,12
   (f) [81], Chp 13.

(g) [144], Chp 2-4
(h) [166, Chp. 9, pp. 393–423; Chp. 12, pp. 562–606; Chp. 13, pp. 677–685]
(i) [174, §3.1; §8.1]
(j) [188], Chp 29
(k) [73], Chp 6,13
(l) [186, pp. 349–359]

11. Function overloading:
    (a) [200], Chp 8
    (b) [82, §6.14, pp. 356–360].
    (c) [83, §6.14, pp. 359–363].
    (d) [81], Chp 6.
    (e) [180, §4.4, pp. 230–243]
    (f) [198], Chp 8
    (g) [166, pp. 216–217, 365–370]. Function overloading is also known as function polymorphism [166, pp. 388].
    (h) [188], Chp 14
    (i) [73], Chp 7
    (j) [186, Chp. 14, pp. 361–384]
12. Operator overloading:
    (a) [128], Chp 14
    (b) [180, §11.2, pp. 633–651]
    (c) [166, pp. 502–515, 524–537]
    (d) [156], Chp 18
    (e) [188], Chp 15
    (f) [187, Chp. 13, pp. 299–330]
    (g) [73], Chp 12
    (h) [186, Chp. 15, pp. 385–418]
13. Constants:
    (a) [73], Chp 8
14. Functions and pointers:
    (a) See [164, §7.2-7.4] regarding passing arguments by value, reference, and by address.
    (b) [200], Chp 8:
        i. Pass-by-reference:
            A. e.g., void init(vector<double> &v)
            B. "It is not possible to refer directly to a reference variable after it is defined; any occurrence of its name refers directly to the variable it references. "
            C. "Once a reference is created, it cannot be later made to reference another variable. This is something that is often done with pointers."
            D. "References cannot be null, whereas pointers can; every reference refers to some variable, although it may or may not be valid."
            E. "References cannot be uninitialized. Because it is impossible to reinitialize a reference, they must be initialized as soon as they are created. In particular, local and global variables must be initialized where they are defined, and references which are data members of a class must be initialized in the initializer list of the class's constructor."

F. Avoid mixing references and pointers in a block of code to avoid confusion, and make it easier for the *C++* code to be read and debug.
G. The required syntax for pointers make them prominent in comparison to that of references.
H. The number of operations on references is less than that on pointers. Hence, usage of references is easier to understand than that of pointers. Consequently, it is easier to use references than pointers without enbugging the code.
I. Pointers can be invalidated as follows:
   - "Carrying a null value"
   - "Out-of-bounds [pointer] arithmetic"
   - Illegal casts on pointers
   - Produce pointers from random integers
J. References can be invalidated as follows:
   - "[Refer] to a variable with automatic allocation which goes out of scope"
   - "[Refer] to an object inside a block of dynamic memory which has been freed"
K. "Arrays are always passed by address. This includes C strings."
L. "Dynamic storage is allocated using pointers."
M. Reference: Kurt McMahon, "Passing Variables by Address," in *Northern Illinois University: College of Engineering and Engineering Technology: Department of Computer Science: CSCI 241 Intermediate Programming in C++ (Fall 2015): Notes*, Northern Illinois University, DeKalb, IL, October 28, 2015. Available online at: http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/pass_by_address.html; last accessed on November 3, 2015.

ii. Pass-by-const-reference: e.g., void print(const vector<double> &v)
iii. Pass-by-value: e.g., void fn(int x)
iv. Pass-by-address: e.g., void print(int * ptr)
A. Reference: Kurt McMahon, "Passing Variables by Address," in *Northern Illinois University: College of Engineering and Engineering Technology: Department of Computer Science: CSCI 241 Intermediate Programming in C++ (Fall 2015): Notes*, Northern Illinois University, DeKalb, IL, October 28, 2015. Available online at: http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/pass_by_address.html; last accessed on November 3, 2015.

(c) [128], Chp 6
(d) [5], Chp 12-13
(e) [101, Chp. 5, pp. 193–248; Chp. 7, pp. 307–349]
(f) [167], Chp 7-8
(g) [180, Chp. 4–5; §11.1; and Chp. 14]
(h) [198], Chp 8
(i) [166, Chp. 7–8, pp. 279–391].:
   i. [166, pp. 382–383] provides information on which function version (from function overloading, function templates, and function template overloading) is chosen by the compiler, during compilation.
   ii. `const` member functions (put the `const` after the function parentheses) guarantees that the invoking object would not be modified.
(j) [156], Chp 15,20
(k) [187, Chp. 6–8, pp. 105–179]
(l) [73], Chp 11:
   i. use const at the end of accessor functions

ii. Do not use pointers as instance variables

(m) [186, Chp. 5–6; pp. 113–160]

(n) Elsewhere:

i. http://stackoverflow.com/questions/1143262/what-is-the-difference-between-const- [151]:

A. Read it backwards; the first *const* can be on either side of the type.
B. "Read pointer declarations right-to-left."
C. From the answer of Ted Dennison, July 17, 2009. **Rule: The "const" goes after the thing it applies to. Putting const at the very front (e.g., const int *) is an exception to the rule.**
D. int* – pointer to int
E. int const * == const int * – pointer to const int
F. int * const – const pointer to int
G. int const * const == const int * const – const pointer to const int
H. int ** – pointer to pointer to int
I. int ** const – A const pointer to a pointer to an int
J. int * const * – A pointer to a const pointer to an int
K. int const ** – A pointer to a pointer to a const int
L. int * const * const – A const pointer to a const pointer to an int

ii. For the following [151], let: *int var0 = 0;*

A. *const int &ptr1 = var0;* // Constant reference
B. *int * const ptr2 = &var0;* // Constant pointer
C. *int const * ptr3 = &var0;* // Pointer to const
D. *const int * const ptr4 = &var0;* // Const pointer to a const

iii. [159]:

A. "A reference is a variable that refers to something else and can be used as an alias for that something else. A pointer is a variable that stores a memory address, for the purpose of acting as an alias to what is stored at that address. So, a pointer is a reference, but a reference is not necessarily a pointer. Pointers are a particular implementation of the concept of a reference, and the term tends to be used only for languages that give you direct access to the memory address. References can be implemented internally in a language using pointers, or using some other mechanism." Answer from dan1111.
B. "Passing an object by value means making a copy of it. You can modify that copy without affecting the original. Making that copy can cost a lot of memory access though. Passing an object by reference means passing a handle to that object. This is cheaper because you don't need to make a copy. It also means that any changes you make will affect the original." Answer from Steve Rowe.
C. "There is no such thing as a null reference. A reference must always refer to some object. As a result, if you have a variable whose purpose is to refer to another object, but it is possible that there might not be an object to refer to, you should make the variable a pointer, because then you can set it to null. On the other hand, if the variable must always refer to an object, i.e., if your design does not allow for the possibility that the variable is null, you should probably make the variable a reference." Answer from Harssh S. Shrivastava.

iv. A pointer is dereferenced via the explicit * operator. The * operator should not be used to dereference a reference (variable) [176].

v. [176]:

A. int *pi = &i; // Indirect expression to dereference *pi* to *i*. "Declare *pi* as an object of type 'pointer to int' whose initial value is the address of object *i*" [177].
B. int &ri = i; // *ri* is dereferenced to refer to *i*. "Declares *ri* as an object of type 'reference to int' referring to *i*" [177].
C. The *C++* standard does not dictate how compilers shall implement references. However, popular compilers tend to implement references as pointers. Therefore, there are no significant advantages of using references or pointers.

vi. [177]:
A. "A valid reference must refer to an object; a pointer need not. A pointer, even a const pointer, can have a null value. A null pointer doesn't point to anything."
B. I can bind a reference to a null pointer, but I cannot dereference a null pointer since it can "produce undefined behavior".

vii. You cannot call a non-const method from a const method. That would 'discard' the const qualifier.:
A. http://stackoverflow.com/questions/2382834/discards-qualifiers-error

viii. Pointer to constant data: *const type* variable;* and *type const * variable;*
A. http://www.cprogramming.com/reference/pointers/const_pointers.html

ix. Pointer with constant memory address: *type * const variable = some-memory-address;*
A. http://www.cprogramming.com/reference/pointers/const_pointers.html

x. Constant data with a constant pointer: *const type * const variable = some-memory-address;* and *type const * const variable = some-memory-address;*
A. http://www.cprogramming.com/reference/pointers/const_pointers.html

(o) With shallow copying, I would only copy the memory references or pointers. The copy and the original reference the same object. On the other hand, with deep copying, I would copy the values; this is also known as cloning. The copy and the original reference do not share objects; each of them references its own object. The default copy constructor carries out shallow copy.

15. Extern function:
(a) :
i.
(b) :
i.
(c) :
i.
(d) :
i.
(e) :
i.
(f) :
i.
(g) :
i.

16. `typename`:
(a) Evan Driscoll, "A Description of the C++ `typename` keyword," from the Department of Computer Sciences, University of Wisconsin-Madison College of Engineering, University of Wisconsin-Madison, Madison, WI. Available online at: http://pages.cs.wisc.edu/~driscoll/typename.html; last accessed on February 15, 2016.

(b) From [10, API documentation for Eigen3: The template and typename keywords in C++], or http://eigen.tuxfamily.org/dox/TopicTemplateKeyword.html.

(c) Wikipedia contributors, "typename," in *Wikipedia, The Free Encyclopedia: C++*, Wikimedia Foundation, San Francisco, CA, April 13, 2015. Available online at: https://en.wikipedia.org/wiki/Typename; last accessed on February 15, 2016.:

i. Usage #1: "A synonym for 'class' in template parameters"

ii. Usage #2: "A method for indicating that a dependent name is a type"

(d) [180, pp. 916]

(e)

(f)

(g)

(h)

(i)

(j)

(k)

17. OOD and inheritance:
    (a) [90], Chp 4-9
    (b) [200], Chp 9
    (c) [128], Chp 7,15,18,19
    (d) [5], Chp 24-26
    (e) [82, Chp. 13–15; Appendices E and J].
    (f) [101, Chp. 9–10, 389–479]
    (g) [167], Chp 10-11,13,14,15
    (h) [83, Chp. 7, 11, 15; Appendices A, D, J, and K].
    (i) [81], Chp 13,14,15.
    (j) [180, Chp. 10; §12.1, 696–711; Chp. 15; Chp. 17]
    (k) [198], Chp 9
    (l) [166, Chp. 10–14, pp. 445–786]
    (m) [156], Chp 13-14,21
    (n) [187, pp. 6–8; Chp. 11–12, pp. 245–298; Chp. 14–15, pp. 331–373]
    (o) [210], Chp 3-4,8
    (p) [73], Chp 14,15
    (q) [186, Chp. 11–12, pp. 255–325; Chp. 16–17, pp. 419–460]
18. SW engineering issues:
    (a) [90], Chp 24-26
    (b) [5], Chp 21
    (c) [174, Chp. 4 and 7]
    (d) Debugging:
        i. [184, §14.2]
19. multi-threading:
    (a) [189], Chp 3
    (b) [174, §5.1]
20. graphs:
    (a) [189], Chp 7
21. typedef:

(a) In the sandbox, use the *Make* target *make typedef* to study an example of how *typedef* can be used. When the *header file* defines/specifies the *typedef*, and is included in the *C++ implementation file* and other *C++ implementation file*s that instantiates those objects, it can be used subsequently without additional definition/specification. October 6, 2015.
(b) [180, pp. 510-512]

I am skipping topics, such as: run-time type ID and casting operators [187, Chp. 19, pp. 451–470]; namespaces [187, pp. 472–480]; #error *C++* preprocessor [187, pp. 552]; #line *C++* preprocessor [187, pp. 558]; #pragma *C++* preprocessor [187, pp. 559]; ## *C++* preprocessor operator [187, pp. 559–560]; and *C++* predefined macro names [187, pp. 560–561].

6.2 Computational Complexity of C++ Containers

Table 6.1 shows a tabulated summary of containers in the *C++* Standard Template Library (STL) and the computational complexity for each of their common operations: add(element e), remove(element e), search(element e), size(), empty(), begin(), and end().

Table 6.1: Computational Complexity of Basic Operations of Containers from the *C++ STL*.

Container \ Complexity	add	remove	search	size	empty	begin	end
vector	O(1)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
list	O(1)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
queue	O(1) amortized	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
priority queue	O(log n)	O(log n)	O(log n)???, or O(n)	O(1)	O(1)	O(1)	???
set	O(log n)	O(log n)	O(log n)	O(1)	O(1)	O(1)	O(1)
multi-set	O(log n)	???	O(log n)	O(1)	O(1)	O(1)	O(1)
map	O(log n)	O(log n)	O(log n)	O(1)	O(1)	O(1)	O(1)
multi-map	O(log n)	???	O(log n)	O(1)	O(1)	O(1)	O(1)
stack	O(1)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)

To conclude, we can get some facts about each data structure:

1. `std::list` is very very slow to iterate through the collection due to its very poor spatial locality.
2. `std::vector` and `std::deque` perform always faster than `std::list` with very small data
3. `std::list` handles very well large elements
4. `std::deque` performs better than a `std::vector` for inserting at random positions (especially at the front, which is constant time)
5. `std::deque` and `std::vector` do not support very well data types with high cost of copy/assignment

This draws simple conclusions on the usage of each data structure [34, 111]:

1. Number crunching: use std::vector or std::deque
2. Linear search: use std::vector or std::deque
3. Random Insert/Remove:
4. Small data size: use std::vector
5. Large element size: use std::list (unless if intended principally for searching)

6. Non-trivial data type: use std::list unless you need the container especially for searching. But for multiple modifications of the container, it will be very slow.
7. Push to front: use std::deque or std::list

Notes about asymptotic notations:

1. Comparison of big O notations, and other asymptotic notations, in general – based on "running time (T(n))" [Wikipedia 2015a][Wikipedia 2015]:
   (a) $O(1)$: constant time
   (b) $O(\log^* n)$, log star: iterated logarithmic time.
   Log star n is a recursive function; $\log^* n := \begin{cases} 0 & : \text{if } n \leq 1 \\ 1 + \log^*(\log n) & : \text{if } n > 1 \end{cases}$ [Wikipedia 2015b]
   (c) $O(\log \log n)$: log-logarithmic, double logarithmic
   (d) $O(\log n)$: logarithmic time, computational time complexity class DLOGTIME. E.g., $\log n^2$.
   (e) $\text{poly}(\log n)$ or $O((\log n)^c), c > 1$: polylogarithmic time. E.g., $(\log n)^2$.
   (f) $O(n^c)$, where $0 < c < 1$: fractional power. E.g., $n^{\frac{2}{3}}$.
   (g) $o(n)$: sub-linear time (or sublinear time)
   (h) $O(n)$: linear time
   (i) $O(n \log^* n)$: "n log star n" time, or "n log-star n"
   (j) $O(n \log n) = O(\log n!)$: linearithmic time, including $\log n!$. Or, loglinear, or quasilinear.
   (k) $O(n^2)$: quadratic time
   (l) $O(n^3)$: cubic time
   (m) $\text{poly}(n)$, or $2^{O(\log n)}$. Or, $O(n^c), c > 1$: polynomial time, including $n, n \log n, n^{10}$. Computational time complexity class P. Or, algebraic.
   (n) $2^{\text{poly}(\log n)}$: quasi-polynomial time, including $n^{\log \log n}, n^{\log n}$. Computational time complexity class QP.
   (o) $O(2^{n^\varepsilon}), \forall \varepsilon > 0$: sub-exponential time, including $O(2^{\log n^{\log \log n}})$. Computational time complexity class SUBEXP.
   (p) $2^{o(n)}$: sub-exponential time, including $2^{n^{\frac{1}{3}}}$. Computational time complexity class SUBEXP. Or, L-notation.
   (q) $2^{O(n)}$: exponential time (with linear exponent), including $1.1^n, 10^n$. Computational time complexity class E.
   (r) $2^{\text{poly}(n)}$. Or, $O(c^n), c > 1$: exponential time, including $2^n, 2^{n^2}$. Computational time complexity class EXPTIME.
   (s) $O(n!)$: factorial time, including $n!$.
   (t) $2^{2^{\text{poly}(n)}}$: double exponential time, including 2^{2^n}. Computational time complexity class 2-EXPTIME.
   (u) $n! > n^n$
2. Types of asymptotic notations [Wikipedia 2015]:
   (a) $f(n) = O(g(n))$: Big O notation, or Big Oh notation
   (b) $f(n) = \Omega(g(n))$: Big Omega notation
   (c) $f(n) = \Theta(g(n))$: Big Theta notation
   (d) $f(n) = o(g(n))$: Small O notation, or Small Oh notation
   (e) $f(n) = \omega(g(n))$: Small Omega notation
   (f) $f(n) \sim g(n)$: "On the order of"
3. References:

(a) [Wikipedia 2015] Wikipedia contributors, "Big O notation," sections *Orders of common functions* and *Related asymptotic notations: Family of Bachmann?Landau notations*, in *Wikipedia, The Free Encyclopedia: Analysis of algorithms, or Asymptotic analysis*, Wikimedia Foundation, San Francisco, CA, November 29, 2015. Available online at: https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions; last accessed on December 1, 2015.

(b) [Wikipedia 2015a] Wikipedia contributors, "Time complexity," section *Table of common time complexities*, in *Wikipedia, The Free Encyclopedia: Computational complexity theory*, Wikimedia Foundation, San Francisco, CA, November 16, 2015. Available online at: https://en.wikipedia.org/wiki/Time_complexity#Table_of_common_time_complexities; last accessed on December 1, 2015.

(c) [Wikipedia 2015b] Wikipedia contributors, "Iterated logarithm," in *Wikipedia, The Free Encyclopedia: Asymptotic analysis*, Wikimedia Foundation, San Francisco, CA, November 6, 2015. Available online at: https://en.wikipedia.org/wiki/Iterated_logarithm; last accessed on December 1, 2015.

4. Note that I denote "is defined as" as: $\equiv, \triangleq, \stackrel{\text{def}}{=}, :=$
5. Note that $\log n$ is faster than $(\log n)^2$, although initially the latter is slightly faster than the former (for negligibly small n).

Books on computational complexity:

1. [113, pp.10–11]

6.3 Additional Notes About C++

Static variables:

1. K. Hong, "Static Variables and Static Class Members - 2015," San Francisco, CA. Available online from *Open Source . . . : Java/C++/Python/Android/Design Patterns: C++ Tutorial Home – 2015* at: http://www.bogotobogo.com/cplusplus/statics.php; last accessed on October 23, 2015.

Formatting data:

1. Synesis Software Pty Ltd staff, "Synesis Software Training Courses: FastFormat, Beginner's (part 1 of 2)," Synesis Software Pty Ltd, Sydney, Australia, 2015. Available online at: http://www.synesis.com.au/training-beginners-fastformat.html; December 1, 2015 was the last accessed date.
   (a) "Formatting APIs":
      i. "Replacement-based APIs":
         A. "Streams (printf()-family)"
         B. "Boost.Format"
         C. "FastFormat.Format"
      ii. "Concatenation-based APIs":
         A. "IOStreams"
         B. "Loki.SafeFormat"
         C. "FastFormat.Write"
   (b) "struct tm"
   (c) "struct in_addr"
   (d) "ATL types"
   (e) "ACE types"

2. Synesis Software Pty Ltd staff, "Synesis Software Training Courses: FastFormat, Advanced (part 2 of 2)," Synesis Software Pty Ltd, Sydney, Australia, 2015. Available online at: http://www.synesis.com.au/training-advanced-fastformat.html; December 1, 2015 was the last accessed date.
   (a) "Format-specification Defect Handling": "Scoping" and "Disgnostic Logging"

6.3.1 Alternate Computer Number System for Representing Fractions in C++

An alternate computer number system for representing fractions in C++ is the fixed-point number system. For a detailed classification of computer number systems, see [132].

In *C++*, the numerical data types are based on cardinal numbers (e.g., one, two, three, ...), instead of ordinal numbers/integers (e.g., first, second, third, ...); see [68, 161, 238] for the definitions of "cardinal number" and "ordinal number." From [161], "a Nominal Number is a number used only as a name, or to identify something (not as an actual value or position)." E.g., "the number on the back of a footballer ("8")," "a postal code ("91210")," and "a model number ("380")."

Resources to help me implement the fixed-point "data type" as a class in *C++*, and fixed-point arithmetic:

1.

6.4 Software Development in C++

Notes about software development in *C++*:

1. Notes from Synesis Software Pty Ltd:
   (a) Synesis Software Pty Ltd staff, "Synesis Software Training Courses," Synesis Software Pty Ltd, Sydney, Australia, 2015. Available online at: http://www.synesis.com.au/training.html; December 1, 2015 was the last accessed date.
      i. Use `FastFormat` as a "C++ diagnostic logging API library"
      ii. `STLSoft libraries`. "Apply the concepts, principles and techniques of Extended STL to enhance the expressiveness, flexibility, and performance of your C++ software." See [237] for more details.
      iii. "Building Bullet-Proof Software in C++ - no system built by Synesis Software has ever failed in production. This course takes you through the principles and practices of how we develop software, providing you with practical, applicable strategies and tactics for achieving the same outcome in your software developments."
      iv. "Guerilla Testing C++ - or, **'How to discover the Gold Nuggets in your Big Ball of Mud'**. No matter how badly a C++ codebase is enmeshed, you can get it under test if you know how to master its coupling."
   (b) Synesis Software Pty Ltd staff, "Resources," Synesis Software Pty Ltd, Sydney, Australia. Available online at: http://www.synesis.com.au/resources.html; December 1, 2015 was the last accessed date.
      i. 100% type-safe *C++* API
      ii. C++ diagnostic logging API library (or, diagnostic logging libraries):
         A. Pantheios: http://pantheios.org/
         B. ACE

C. log4cxx

iii. C++ formatting library: FastFormat http://fastformat.org/

iv. "The STLSoft libraries provide STL extensions and facades over operating-system and third-party-library APIs. The libraries are 100% header-only." See http://stlsoft.org/.

v. "UNIXem is a simple library that emulates a useful subset of the UNIX system APIs on Windows... UNIXem is the only library provided by Synesis Software that is not production-quality. It is appropriate for research, such as when developing tests for cross-platform software." See http://synesis.com.au/software/unixem.html.

(c) Synesis Software Pty Ltd staff, "Guerilla Testing C++ or, 'How to discover the Gold Nuggets in your Big Ball of Mud'," Synesis Software Pty Ltd, Sydney, Australia. Available online at: http://www.synesis.com.au/training-guerilla-testing-cplusplus.html; December 1, 2015 was the last accessed date.:

i. "Change is the most expensive part of the cost of a software project. The biggest impediments to change are lack of clarity on what to alter to effect the change, and uncertainty about unintended side-effects of the change."

ii. "No matter how badly a C++ codebase is enmeshed, you can get it under test if you know how to master its coupling."

iii. "Many long-lived codebases have evolved to a point where some, perhaps most, aspects of its functionality are no longer precisely known / codified / automatically tested. This course will teach you, using practical examples, how to wrest control from any codebase, no matter how badly enmeshed, isolate known pieces of good functionality, get them under test, and eventually to isolate and separate them into a new context, while, where required, maintaining compatibility with their original context."

iv. "This course will teach you how to refactor any codebase with confidence, rather than poking at the edges of its functionality in fear."

v. "Release costs" serve as an indicator to the existence of "a Big Ball of Mud."

vi. "Factors that inhibit testing":

A. "Coupling, coupling, coupling"

B. "The inconstant environment"

C. "Trust"

D. "Defensive code"

E. "Fuzzy (or no!) abstraction borders"

vii. "Key characteristics" identified in situ: "diagnostics, contracts, code coverage, and testing."

viii. Remember the following "when testing mud-balls": "automation; minimalism, incrementality, unit testing vs component testing; coverage (in realistic time); only change what you can test (and are testing!) – [there are] exceptions to this rule; beyond salvation – sometimes it's just mud."

ix. Islands of "known Functionality" are created as follows:

A. "Decomposition – Identifying Units, Identifying Components, and Identifying Modules"

B. "Triage"

C. "Isolation"

D. "Striding two worlds"

E. "Transplantation"

F. "Separation"

G. "Versioning – Static and Dynamic"

H. "When to 'throw it out'."

x. "Inconstant Environment" handling:

A. "File system"
B. "Memory"
C. "User-interface"
D. "Time"
E. "Data storage"

xi. Techniques to address/mitigate coupling:
A. "Pre-processor":
- #ifdef
- #define
- #include
- "I would recommend putting your include guards above your includes. That way the includes don't get parsed twice for the same file." That is, the pre-processors `#ifdef` and `#define` should be placed above the `#include` statements. This would avoid repetitive parsing of (header) files that are included. Reference: sdsmith, comment to the question "Why is the discrepancy in these two cases of using C++ templates?," Stack Exchange Inc., New York, NY, March 30, 2016. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: `http://stackoverflow.com/q/36319028/1531728`; March 30, 2016 was the last accessed date [190].

B. "linkage":
- "interpositioning"
- "dynamic library redirection"

C. "object-oriented techniques":
- "overloading"
- "overriding"
- "inheritance"
- "interfaces"

D. "patterns":
- "class adaptor"
- "instance adaptor"
- "decorator"
- "visitor"

E. "generic programming":
- "policies"
- "shims"
- "traits"

F. "Testing":
- "Stubbing"
- "Mocking"
- "Versioned testing"

xii.
xiii.
xiv.
xv.
xvi.
xvii.
xviii.
xix.

2. To obtain the `x86` assembly output to a *C++* program:

(a) Reference: Andrew Edgecombe, answer to "How do you get assembler output from C/C++ source in gcc?," Stack Exchange Inc., New York, NY, September 26, 2008 (edited by Prashant Kumar on May 1, 2012). Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/a/137074; March 15, 2016 was the last accessed date.:

i. To view the source file: `cat` *[source_file]*
ii. To remove all symbol table and relocation information from the executable, the executable for that source file becomes: `gcc -s` *[source_file.c]*:
   A. Or, try: `gcc -S` *[source_file.c]*
   B. Alternatively, try: `strip` *[source_file]*
   C. `gcc -g` *[source_file.c]* adds debugging information to the executable.
iii. If access to source files is unavailable, but access to the object file is available, use: `objdump -S -disassemble` *[object_file]* > *[object_file.dump]*. Use `file` *[object_file]* to determine if I can get enough information from the object file.:
   A. `gcc -S -o` *[unassembled_file.s]* *[source_file.c]*
   B. To view the output file from `gcc -S`: `cat` *[source_file]*

(b) References: [7, pp. 3, or pp. 15 in the PDF] and Kenneth Finnegan, answer to "How do you get assembler output from C/C++ source in gcc?," Stack Exchange Inc., New York, NY, September 26, 2008. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/a/137479; March 16, 2016 was the last accessed date. Also, includes information from comments to this answer by Sundaram Ramaswamy (May 6, 2013) and Lưu Vĩnh Phúc (June 7, 2014):

i. Create assembly code: `c++ -S -fverbose-asm -g -O2` *[source_code.cpp]* `-o` *[unassembled_file.s]*
ii. Create assembled code interfaced with source lines: `as -alhnd` *[unassembled_file.s]* > *[listing_file.lst]*
iii. Alternatively, the on line version on *Mac OS X*, try:
   A. `g++ -g -O0 -c -fverbose-asm -Wa,-adhln test.cpp > test.lst`
   B. `gcc -c -g -Wa,-ahl=test.s test.c`
   C. `gcc -c -g -Wa,-a,-ad [`*other GCC options*`] test.c > test.txt`

(c) Reference: Cr McDonough, answer to "How do you get assembler output from C/C++ source in gcc?," Stack Exchange Inc., New York, NY, September 29, 2013. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/a/19083877; March 16, 2016 was the last accessed date.

i. `g++ -g -O -Wa,-aslh horton_ex2_05.cpp > list.txt`

(d) Reference: Andrew Pennebaker, answer to "How do you get assembler output from C/C++ source in gcc?," Stack Exchange Inc., New York, NY, February 6, 2013. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/a/7871911; March 16, 2016 was the last accessed date. Also, comment from Grady Player (February 6, 2013) is helpful.

i. To get the LLVM assembly code, try: `llvm-gcc -emit-llvm -S hello.c`
ii. I can also use the same command for `clang`.

6.4.1 Using "Design By Contract"

The "Design By Contract" approach shall be used in software development. This approach is also known as: "contract programming, programming by contract, and design-by-contract programming." Adhere strongly to Hoare logic.

References:
Wikipedia contributors, "Design by contract," in *Wikipedia, The Free Encyclopedia: Software design*, Wikimedia Foundation, San Francisco, CA, January 20, 2016. Available online at: https://en.wikipedia.org/wiki/Design_by_contract; last accessed on February 9, 2016.

Wikipedia contributors, "Hoare logic," in *Wikipedia, The Free Encyclopedia: Static program analysis*, Wikimedia Foundation, San Francisco, CA, November 8, 2015. Available online at: https://en.wikipedia.org/wiki/Hoare_logic; last accessed on February 10, 2016.

That is, at the start of an implementation of a (*C++*) function, check that its precondition(s) is (/are) met; preconditions shall be chosen to be as weak as possible. Within the implementation of the function, check if the assertions (properties that must be true during execution of the function) hold. Lastly, at the end of the implementation, check that its postcondition(s) is (/are) met; postconditions shall be chosen to be as strong as possible.

Reference:
Wikipedia contributors, "Predicate transformer semantics," in *Wikipedia, The Free Encyclopedia: Formal methods*, Wikimedia Foundation, San Francisco, CA, November 25, 2015. Available online at: https://en.wikipedia.org/wiki/Predicate_transformer_semantics; last accessed on February 10, 2016.

6.4.1.1 Hoare Logic for Computer Arithmetic

Check if arithmetic and logical operations cause overflows or underflows [52, 53]. When faced with a constrained range for representing real numbers with bits in computer hardware, a constrained resolution such that a representation smaller than single-precision floating-point numbers is required, or low-cost and/or low-power electronic/computer systems that do not have floating-point arithmetic circuits, use a circuit-based implementation of fixed-point arithmetic.

6.4.2 Debugging *C++* Software

This section covers debugging syntax errors (in §6.4.2.1) that are reported by *C++* compilers. It also covers semantic errors that can be discovered via software testing and formal verification tools; see [64, §4.4, pp. 119]. In addition, it includes performance debugging [64, §4.5, pp. 119–120], using software profilers [78, Figure 7.1, pp. 292; and §7.2.10, pp. 302] [117, pp. 148] [126, pp. 35-9 – 35-10] [79, §3.2, pp. 16–18] [11, §3.3.2, pp. 21–22; pp. 23; Figure 40, pp. 102; Appendix C, §C.1.7, pp. 104] and static analysis software [11, §5.4, 65-66; and Figure 40, pp. 102] [3] [12, §7.1, 176–183] [59, §5.2.4, pp. 82–83; and §5.4.2, pp. 90–92] [80, Chapter 7, pp. 109–117] [126, §35.5, pp. 35-10 – 35-14] [28]. These phases cannot be highly overlapped consider by more than a significant amount.

To functionally debug software with success, these four steps should be carried out: "test input generation, error detection, error diagnosis, and error correction" [118].

6.4.2.1 Interpreting *C++* Compilation Errors

A missing semiconlon ";", or lack of matching curly braces "}" (or parentheses ")", square brackets "]", or angle brackets ">"), can cause a lot of compilation errors [103].

6.4.3 Parser Development

Parser development via *Lex/Yacc*, *Flex/Bison*, *ANTLR*, *Parsec*, *Ragel*, *Spirit Parser Framework*, *JetPAG (Jet Parser Auto-Generator)*, *Monkey*, *MyParser*, *SableCC*, ...

Reference: Wikipedia contributors, "Comparison of parser generators," in *Wikipedia, The Free Encyclopedia: Parser generators*, Wikimedia Foundation, San Francisco, CA, February 18, 2016. Available online at: https://en.wikipedia.org/wiki/Comparison_of_parser_generators; last accessed on February 18, 2016.

Determine if `LLVM` *can be used for parser development.*
An example of a *C++* parser is shown in [186, Chp. 40, pp. 959–993].

6.5 Parallel Programming in C++

Resources for parallel programming in C++:

1. C++ -based MPI programming: [115]

6.6 Numerical Computing in C++

Numerical computing resources for *C++*:

1. Scientific computing: [162]
2.

Chapter 7

Questions

7.1 Unresolved C++ Questions

Questions about `C++`:

1.

7.2 Resolved C++ Questions

Difference between pointers and references:

1. Yusuf Kemal Özcan ("BFaceCoder"), "Is there any difference between pointers and references? [duplicate]," Stack Exchange Inc., New York, NY, April 18, 2013. Available online from *Stack Exchange Inc.: Programmers Stack Exchange: Questions* at: http://programmers.stackexchange.com/questions/195337/is-there-any-difference-between-pointers-and-references; October 6, 2015 was the last accessed date.
   (a) Answer from *dan1111*, April 18, 2013: http://programmers.stackexchange.com/a/195343 and http://programmers.stackexchange.com/questions/195337/is-there-any-difference-be 195343#195343.
   (b)
2. Macneil Shonle and Programmers Stack Exchange contributors, "What's a nice explanation for pointers? [closed]," Stack Exchange Inc., New York, NY, July 30, 2015. Available online from *Stack Exchange Inc.: Programmers Stack Exchange: Questions* at: http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanation-for-pointers; October 6, 2015 was the last accessed date.
   (a) Answer from Kevin, November 10, 2010: http://programmers.stackexchange.com/a/17919 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanation-fo 17919#17919. "A pointer is a variable that contains an address to a variable. A pointer is both defined and dereferenced (yielding the value stored at the memory location that it points to) with the '*' operator; the expression is mnemonic." ... `char (*(x())[])()`
   (b) Answer from Barfield, November 10, 2010: http://programmers.stackexchange.com/a/18087 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanati 18087#18087. "Pointer[s] are a bit like the application shortcuts on your desktop."
   (c) Answer from Gulshan, November 10, 2010: http://programmers.stackexchange.com/a/17915 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanati 17915#17915. Pointers point to instance and static variables. A pointer can point to different variables during the execution of the program, but must point to one variable at

any instance (i.e., point in time) during execution. Also, the pointer must point to variables of the same type. Associate a pointer with a variable via the reference to the variable; e.g., `int *pointer; pointer = & variable; ...` According to *Ptolemy*, December 2, 2010: http://programmers.stackexchange.com/a/23016. `int *pointer = & variable;` creates a pointer to the variable. ... Dereference the pointer (add * as a prefix) to store the value of an expression (based on variables, strings, or constants). According to *Ptolemy*, `& variable` is the "address of the variable" and it "represents the literal value for" the pointer. "The pointer" refers to the data that the pointer points to, or something "pointed to by" the pointer.

(d) Answer from Sridhar Iyer, November 11, 2010: http://programmers.stackexchange.com/a/18529 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explana 18529#18529. A "pointer is a variable that store[s] the address of another variable (or just any variable). * is used to get the value at the memory location that is stored in the pointer variable. & operator gives the address of a memory location."

(e) Answer from *rwong*, November 2, 2010: http://programmers.stackexchange.com/a/18054 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanation-fo 18054#18054. Each pointer, which is a special type of variable, must point to only one variable. Variables that are not pointers must not point to anything; however, such variables can be pointed to by any number of pointers.

(f) Answer from *back2dos*, November 10, 2010: http://programmers.stackexchange.com/a/18092 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanati 18092#18092. The pointer [variable] interprets the value of the pointer [variable] as the address of another variable that it points to. Hence, the value of the pointer [variable] refers to a specific location in memory (specified by the address), and is called the reference. Dereferencing is the process of accessing the value of the memory location that it points/refers to. That is, $^*v$ dereferences the value of v, and provides the value at the memory location referred to by the address in v. $\&v$ provides a reference (or the address of the memory location for v) to the variable v.

(g) Answer from *Ptolemy*, December 2, 2010: http://programmers.stackexchange.com/a/23016 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanati 23016#23016. At a low level, the concept of memory can be viewed as a massive array. "Any position in the array" can be accessed "by its index location." "Passing the index location rather than copying the entire memory" is more efficient in terms of performance and memory usage. Hence, "pointers are useful." "For [a] method to store the index location [of] where all the data [in the array] is stored," "a memory index location" can be passed in as a parameter. Pointers can be chained indefinitely; "keep track of how many times [I] need to look at the addresses to find the actual data object." While pointers to heap memory are safe, "pointers to stack memory are dangerous when passed outside the method."

(h) Also, see http://www.udel.edu/CIS/105/pconrad/03F/2003.fall.doc by "P. Conrad."

3. [109, pp. 15, second last paragraph]
   (a) "The value of a pointer is the address to which it points"; or, the "the value of a pointer is the address."

4. [77]
   (a) "pointers use the * and $->$ operators, references use ."
   (b) "Both pointers and references let you refer to other objects indirectly."
   (c) "there is no such thing as a null reference"
   (d) "A reference must always refer to some object."

(e) **"As a result, if you have a variable whose purpose is to refer to another object, but it is possible that there might not be an object to refer to, you should make the variable a pointer, because then you can set it to null."**

(f) *"On the other hand, if the variable must always refer to an object, i.e., if your design does not allow for the possibility that the variable is null, you should probably make the variable a reference."*

(g) "Because a reference must refer to an object, C++ requires that references be initialized." ... Pointers do not have to be initialized; i.e., pointers can be uninitialized. However, "uninitialized pointers" are "valid but risky."

(h) Since null references do not exist, references can be used more efficiently than pointers. This is because the validity of a reference does not have to be tested prior to usage.

(i) Before using pointers, they should be tested against null (i.e., check the validity of a reference prior to usage).

(j) "Pointers may be reassigned to refer to different objects." "A reference ... always refer to the object with which it is initialized."

(k) "You should use a pointer whenever you need to take into account the possibility that there's nothing to refer to (in which case you can set the pointer to null) or whenever you need to be able to refer to different things at different times (in which case you can change where the pointer points)."

(l) "You should use a reference whenever you know there will always be an object to refer to and you also know that once you're referring to that object, you'll never want to refer to anything else."

(m) "There is one other situation in which you should use a reference, and that's when you're implementing certain operators. The most common example is operator[]. This operator typically needs to return something that can be used as the target of an assignment."

(n) "References, then, are the feature of choice when you know you have something to refer to, when you'll never want to refer to anything else, and when implementing operators whose syntactic requirements make the use of pointers undesirable. In all other cases, stick with pointers."

5. Prakash Rajendran, Theodore Logan (Commodore Jaeger), Josh Lee, sbi, Rob$_{\varphi}$, Sudhanshu Aggarwal, lpapp, Alf, Deduplicator, Sam, and Siddhant Saraf, "What are the differences between a pointer variable and a reference variable in C++?," Stack Exchange Inc., New York, NY, March 2, 2015. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/questions/57483/what-are-the-differences-between-a-pointer-variable-a October 8, 2015 was the last accessed date.

   (a) A pointer can be re-assigned any number of times while a reference can not be re-seated after binding.

   (b) Pointers can point nowhere (NULL), whereas reference always refer to an object.

   (c) You can't take the address of a reference like you can with pointers.

   (d) There's no "reference arithmetics" (but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in &obj + 5).

   (e) Use references in function parameters and return types to define useful and self-documenting interfaces.

   (f) Use pointers to implement algorithms and data structures.

6. (a)
7. (a)
8. (a)
9. (a)
10. (a)
11. (a)

12. (a)
13. (a)
14. (a)
15. (a)
16. (a)

Chapter 8

Miscellaneous

List of interesting resources:

1. List of important publications in theoretical computer science: https://en.wikipedia.org/wiki/List_of_important_publications_in_theoretical_computer_science
2. List of important publications in computer science: https://en.wikipedia.org/wiki/List_of_important_publications_in_computer_science
3. List of important publications in mathematics: https://en.wikipedia.org/wiki/List_of_important_publications_in_mathematics#Graph_theory

8.1 Setting Up Software Development Environment

Setting up software development environment for `C++`:

1. Platform-independent environments and software:
   (a) *Linux*, *Mac OS X*, and *Microsoft Windows*:
      i.
   (b) Truly platform independent:
      i.
2. *Mac OS X*:
   (a) Integrated development environments (IDEs):
      i. *Xcode*:
         A. *Preferences* $\Longrightarrow$ *Text Editing* $\Longrightarrow$ *Editing* $\Longrightarrow$ *Code folding ribbon*
         B. *Preferences* $\Longrightarrow$ *Text Editing* $\Longrightarrow$ *Indentation* $\Longrightarrow$ *Syntax-aware indenting: Automatically indent based on syntax + Indent "//" comments one level deeper + Align consecutive "//" comments*
3. *Linux*:
   (a) Text editors:
      i. *gedit*:
         A.
      ii. *NEdit*:
         A.
   (b)

8.2 Software Dependencies of The Boilerplate Code Project

The software dependencies of the "Boilerplate Code" project are found in the following *Markdown* file: `.../lamiera-per-caldaie/notes/miscellaneo/software-dependencies.md`.

8.3 Food for Thought

In [33]:

1. It mentions the following quote from the reference/book (indicated below) by Klir and Marin: "The most powerful tool of the modern methodology of switching circuits seems to be the Boolean equations. Their importance for switching theory reminds one of the application of differential equations in electric circuit theory.":
   (a) It also mentions the following: "There remains a curious difference, however, between the way differential equations and Boolean equations are typically applied: Boolean equations are rarely solved. They are manipulated in form but are seldom the basis for systematic reasoning. Contemporary research on Boolean methods in switching tends instead to emphasize formula-minimization."
   (b) Reference mentioned in [33]: Klir, G.J. and M.A. Marin, "New considerations in teaching switching theory," IEEE Trans. on Education, vol. E-12, pp. 257-261, 1969.
2. It quotes the following from the given reference.
   (a) "A widely-used text [84, p. 60] announces that "Almost every problem in Boolean algebra will be found to be some variation of the following statement: 'Given one of the 2^{2^n} functions of n variables, determine from the large number of equivalent expressions of this function one which satisfies some criteria for simplicity.' " Boole would doubtless deem that to be less than full employment for the algebra he designed as an instrument for reasoning."
   (b) Hill, F.J. and G.R. Peterson, Switching Theory and Logical Design, Third Edition. New York: Wiley, 1981.

Acknowledgments

I would like to thank Ms. Deepika Panchalingam for motivating me to revise basic data structures and algorithms for internship and job interviews.

I would also like to thank the following users on `Stack Overflow` for their help: sdsmith2016 [190].

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. The C++ In-Depth Series. Pearson Education, Boston, MA, 2005.

[2] Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, E. A. Boyd, Sebastian Ceria, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, R. R. Indovina, Thorsten Koch, Andrea Lodi, Alexander Martin, Cassandra M. McZeal, Hans Mittelmann, Ted Ralphs, Daniel Rehfeldt, Domenico Salvagnin, Martin W. P. Savelsbergh, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010 bibliography. Available online from *Konrad-Zuse-Zentrum für Informationstechnik Berlin: MIPLIB – Mixed Integer Problem Library* at: http://miplib.zib.de/biblio.html; December 9, 2015 was the last accessed date, December 1 2015.

[3] Sarita V. Adve, Doug Burger, Rudolf Eigenmann, Alasdair Rawsthorne, Michael D. Smith, Catherine H. Gebotys, Mahmut T. Kandemir, David J. Lilja, Alok N. Choudbary, Jesse Z. Fang, and Pen-Chung Yew. Changing interaction of compiler and architecture. *IEEE Computer*, 30(12):51–58, December 1997.

[4] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. The C++ In-Depth Series. Addison-Wesley, Indianapolis, IN, 2001.

[5] Alex Allain. *Jumping into C++*. Cprogramming.com, San Francisco, CA, 2012.

[6] Charles J. Alpert. *Multi-way Graph and Hypergraph Partitioning*. PhD thesis, University of California, Los Angeles, Los Angeles, CA, July 1996.

[7] Jörg Arndt. Matters computational: Ideas, algorithms, source code. Available online at: http://www.jjj.de/fxt/ and http://www.jjj.de/fxt/fxtbook.pdf; self-published; October 9, 2014 was the last accessed date, September 7 2010.

[8] Mikhail J. Atallah and Marina Blanton. *Algorithms and Theory of Computation Handbook: General Concepts and Techniques*, volume 1 of *Chapman & Hall/CRC Applied Algorithms and Data Structures*. CRC Press, Boca Raton, FL, second edition, 2009.

[9] Mikhail J. Atallah and Marina Blanton. *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*, volume 2 of *Chapman & Hall/CRC Applied Algorithms and Data Structures*. CRC Press, Boca Raton, FL, second edition, 2009.

[10] Philip Avery, Abraham Bachrach, Sebastien Barthelemy, Carlos Becker, David Benjamin, Cyrille Berger, Armin Berres, Jose Luis Blanco, Mark Borgerding, Romain Bossart, Kolja Brix, Gauthier Brun, Philipp Büttgenbach, Thomas Capricelli, Nicolas Carre, Jean Ceccato, Vladimir Chalupecky, Benjamin Chrétien, Andrew Coles, Jeff "complexzeros", Marton Danoczy, Jeff Dean, Georg

Drenkhahn, Christian Ehrlicher, Martinho Fernandes, Daniel Gomez Ferro, Rohit Garg, Mathieu Gautier, Anton Gladky, Stuart Glaser, Marc Glisse, Frederic Gosselin, Gaël Guennebaud, Philippe Hamelin, Marcus D. Hanwell, David Harmon, Chen-Pang He, Hauke Heibel, Christoph Hertzberg, Pavel Holoborodko, Tim Holy, Intel staff, Trevor Irons, Benoît Jacob, Bram de Jong, Kibeom Kim, Moritz Klammler, Claas Köhler, Alexey Korepanov, Igor Krivenko, Marijn Kruisselbrink, Abhijit Kundu, Moritz Lenz, Bo Li, Sebastian Lipponer, Daniel Lowenberg, David J. Luitz, Naumov Maks, Angelos Mantzaflaris, D. J. Marcin, Konstantinos A. Margaritis, Roger Martin, Ricard Marxer, Vincenzo Di Massa, Christian Mayer, Frank Meier-Dörnberg, Keir Mierle, Laurent Montel, Eamon Nerbonne, Alexander Neundorf, Jason Newton, Jitse Niesen, Desire Nuentsa, Jan Oberländer, Jos van den Oever, Michael Olbrich, Simon Pilgrim, Bjorn Piltz, Benjamin Piwowarski, Zach Ploskey, Giacomo Po, Sergey Popov, Manoj Rajagopalan, Stjepan Rajko, Jure Repinc, Kenneth Frank Riddile, Richard Roberts, Adolfo Rodriguez, Peter Román, Oliver Ruepp, Radu Bogdan Rusu, Guillaume Saupin, Olivier Saut, Benjamin Schindler, Michael Schmidt, Dennis Schridde, Jakob Schwendner, Christian Seiler, Martin Senst, Sameer Sheorey, Andy Somerville, Alex Stapleton, Benoit Steiner, Sven Strothoff, Leszek Swirski, Adam Szalkowski, Silvio Traversaro, Piotr Trojanek, Anthony Truchet, Adolfo Rodriguez Tsourouksdissian, James Richard Tyrer, Rhys Ulerich, Henry de Valence, Ingmar Vanhassel, Michiel Van Dyck, Scott Wheeler, Freddie Witherden, Urs Wolfer, Manuel Yguel, and Pierre Zoppitelli. Eigen. Available online at: http://eigen.tuxfamily.org/index.php?title=Main_Page; February 15, 2016 was the last accessed date, February 12 2016.

[11] Hyunki Baik, François Bodin, Ross Dickson, Max Domeika, Scott A. Hissam, Skip Hovsmith, James Ivers, Ian Lintault, Stephen Olsen, and David Stewart. Multicore programming practices guide. Technical report, The Multicore Association, El Dorado Hills, CA, 2013.

[12] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System-Level Methodology*. The Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann, San Francisco, CA, 2007.

[13] Jørgen Bang-Jensen and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer Monographs in Mathematics. Springer-Verlag London, London, U.K., second edition, 2009.

[14] R. B. Bapat. *Graphs and Matrices*. Universitext. Hindustan Book Agency and Springer-Verlag London, New Delhi, India and London, U.K., 2010.

[15] Ravindra B. Bapat. *Graphs and Matrices*. Universitext. Hindustan Book Agency and Springer-Verlag London, New Delhi, India and London, U.K., second edition, 2014.

[16] David Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, New York, NY, 2012.

[17] Gerassimos Barlas. *Multicore and GPU Programming: An Integrated Approach*. Morgan Kaufmann, Waltham, MA, 2015.

[18] Wojciech Basalaj and Richard Corden. High integrity C++ coding standard V4.0: An overview. Technical report, Programming Research Ltd., Hersham, Surrey, England, U.K., November 2013.

[19] Amit Basu and Robert W. Blanning. *Metagraphs and Their Applications*, volume 15 of *Integrated Series in Information Systems*. Springer Science+Business Media, LCC, New York, NY, 2017.

[20] Pete Becker. Working draft, standard for programming language C++. Technical Report N3242, International Organization for Standardization and International Electrotechnical Commission, Genève, Switzerland, February 28 2011.

[21] Ben. answer to the question 'why can templates only be implemented in the header file?'. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/a/16493574/1531728; March 30, 2016 was the last accessed date, May 11 2013.

[22] Benoît. answer to the question 'why can templates only be implemented in the header file?'. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/a/495128/1531728; edited by Evan Teran, February 10, 2009; March 31, 2016 was the last accessed date, January 30 2009.

[23] Claude Berge. *Hypergraphs: Combinatorics of Finite Sets*, volume 45 of *North-Holland Mathematical Library*. North-Holland, Amsterdam, The Netherlands, 1989.

[24] Dimitris Bertsimas and Melvyn Sim. The price of robustness. *Operations Research*, 52(1):35–53, January–February 2004.

[25] Norman Biggs. *Algebraic Graph Theory*, volume 67 of *Cambridge Tracts in Mathematics*. Cambridge University Press, Cambridge, U.K., 1974.

[26] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer Science+Business Media, LCC, New York, NY, 2006.

[27] Robert E. Bixby and Donald K. Wagner. An almost linear-time algorithm for graph realization. *Mathematics of Operations Research*, 13(1):99–123, February 1988.

[28] Jean-Louis Boulanger. *Static Analysis of Software: The Abstract Interpretation*. John Wiley & Sons, Hoboken, NJ, 2013.

[29] Alain Bretto. *Hypergraph Theory: An Introduction*. Mathematical Engineering. Springer International Publishing Switzerland, Cham, Switzerland, 2013.

[30] Gary Bronson. *A First Book of C++*. Course Technology, Boston, MA, fourth edition, 2012.

[31] Gary J. Bronson. *C++ for Engineers and Scientists*. Cengage Learning, Boston, MA, fourth edition, 2013.

[32] Andries E. Brouwer and Willem H. Haemers. *Spectra of Graphs*. Universitext. Springer Science+Business Media, LCC, New York, NY, 2012.

[33] Frank Markham Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Dover Publications, Mineola, NY, second edition, 2003.

[34] Dov Bulka and David Mayhew. *Efficient C++: Performance Programming Techniques*. Addison Wesley Longman, Inc., Indianapolis, IN, 2000.

[35] Horst Bunke, Abraham Kandel, and Mark Last. *Applied Pattern Recognition*, volume 91 of *Studies in Computational Intelligence*. Springer-Verlag Berlin Heidelberg, Heidelberg, Germany, 2008.

[36] Frank M. Carrano and Timothy Henry. *Data Abstraction & Problem Solving with C++: Walls and Mirrors*. Prentice Hall, Upper Saddle River, NJ, sixth edition, 2013.

[37] Peter J. Carrington, John Scott, and Stanley Wasserman. *Models and Methods in Social Network Analysis*, volume 27 of *Structural Analysis in the Social Sciences*. Cambridge University Press, New York, NY, 2005.

[38] Sankar Deep Chakraborty. *Space efficient graph algorithms*. PhD thesis, Homi Bhabha National Institute, Mumbai, India, March 16 2018.

[39] Gary Chartrand and Ping Zhang. *Chromatic Graph Theory*. Discrete Mathematics and Its Applications. CRC Press, Boca Raton, FL, 2009.

[40] Marshall Cline. C++ FAQ Lite: Frequently asked questions. Available online from the course web page of *CS210 Data Structures and Abstractions Lab*(Spring 2015), Department of Computer Science, Faculty of Science, University of Regina at: http://www.cs.uregina.ca/Links/class-info/210/C++FAQ/; self-published; July 10, 2015 was the last accessed date, July 10 2000.

[41] Marshall Cline. C++ FAQ Lite: Frequently asked questions. Available online from the Computer Science Department, B. Thomas Golisano College of Computing and Information Sciences, Rochester Institute of Technology at: http://www.cs.rit.edu/~mjh/docs/c++-faq/; self-published; July 10, 2015 was the last accessed date, May 2 2003.

[42] Marshall Cline. C++ FAQ Lite: Frequently asked questions. Available online from the web page of Laura Mensi and Paolo Copello, *Tiscali Italia S.p.A.: Tiscali Webspace: Fanelia Italy – Computer Programming, Psychiatry, Escaflowne, and much more* at: http://web.tiscali.it/fanelia/cpp-faq-en/; self-published; July 10, 2015 was the last accessed date, July 28 2011.

[43] Jason Cong and Joseph R. Shinnerl. *Multilevel Optimization in VLSICAD*, volume 14 of *Combinatorial Optimization*. Kluwer Academic Publishers, New York, NY, 2003.

[44] Thomas H. Cormen. *Algorithms Unlocked*. The MIT Press, Cambridge, MA, 2013.

[45] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, third edition, 2009.

[46] cplusplus.com. The C++ resources network. Available online at: http://www.cplusplus.com/; April 2, 2014 was the last accessed date, 2014.

[47] cplusplus.com. C++ language. Available online from *The C++ Resources Network: Tutorials* at: http://www.cplusplus.com/doc/tutorial/; February 5, 2016 was the last accessed date, 2015.

[48] cplusplus.com. Reference: C++ reference. Available online at: http://www.cplusplus.com/reference/; November 2, 2015 was the last accessed date, 2015.

[49] cplusplus.com. Tutorials. Available online from *The C++ Resources Network* at: http://www.cplusplus.com/doc/; February 5, 2016 was the last accessed date, 2015.

[50] cppreference.com contributors. C++ reference. Available online at: http://en.cppreference.com/w/cpp; September 17, 2015 was the last accessed date, June 16 2015.

[51] Yves Crama and Peter L. Hammer. *Boolean Functions: Theory, Algorithms, and Applications*, volume 142 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, New York, NY, 2011.

[52] Lawrence Crowl. C++ binary Fixed-Point arithmetic. Technical Report N3352, International Organization for Standardization and International Electrotechnical Commission, Genève, Switzerland, January 15 2012.

[53] Lawrence Crowl. C++ binary Fixed-Point arithmetic. Technical Report P0106R0, International Organization for Standardization and International Electrotechnical Commission, Genève, Switzerland, September 27 2015.

[54] Dragoš Cvetković, Peter Rowlinson, and Slobodan Simić. *An Introduction to the Theory of Graph Spectra*, volume 75 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, U.K., 2010.

[55] Nell Dale. *C++ Plus Data Structures*. Jones and Bartlett Learning, Burlington, MA, fifth edition, 2013.

[56] Nell Dale and Chip Weems. *Programming and Problem Solving with C++*. Jones and Bartlett Learning, Burlington, MA, sixth (comprehensive) edition, 2014.

[57] Michael Dawson. *Beginning C++ Through Game Programming*. Course Technology, Boston, MA, third edition, 2011.

[58] Michael Dawson. *Beginning C++ Through Game Programming*. Course Technology, Boston, MA, fourth edition, 2014.

[59] Mourad Debbabi, Fawzi Hassaïne, Yosr Jarraya, Andrei Soeanu, and Luay Alawneh. *Verification and Validation in Systems Engineering: Assessing UML/SysML Design Models*. Springer-Verlag Berlin Heidelberg, Heidelberg, Germany, 2010.

[60] Paul Deitel and Harvey Deitel. *C++ How to Program*. Deitel^{o} How to Program. Prentice Hall, Boston, MA, eighth edition, 2012.

[61] Paul Deitel and Harvey Deitel. *C++ How to Program*. Deitel^{o} How to Program. Prentice Hall, Upper Saddle River, NJ, ninth edition, 2014.

[62] Paul Deitel and Harvey Deitel. *C++ How to Program*. Deitel^{o} How to Program. Pearson Education, Hoboken, NJ, tenth edition, 2017.

[63] DevSolar. answer to the question 'why can templates only be implemented in the header file?'. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/a/495511/1531728; March 30, 2016 was the last accessed date, January 30 2009.

[64] Ian G. Harris Dhiraj K. Pradhan. *Practical Design Verification*. Cambridge University Press, New York, NY, 2009.

[65] Davide Di Gennaro. Advanced C++ metaprogramming. Self-published, 2011.

[66] Davide Di Gennaro. *Advanced Metaprogramming in Classic C++*. The Expert's Voice^{o} in C++. Apress Media, LLC, Berkeley, CA, 2015.

[67] Germán Diago. answer to the question 'why can templates only be implemented in the header file?'. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/a/16509701/1531728; March 31, 2016 was the last accessed date, May 12 2013.

[68] Dictionary.com staff. Dictionary.com: Find the meanings and definitions of words at dictionary.com. Available online at: http://dictionary.reference.com/; February 5, 2016 was the last accessed date, 2016.

[69] Jack Dongarra and Eric Grosse. Netlib repository. Available online at: http://www.netlib.org/; February 3, 2016 was the last accessed date, 2016.

[70] Allen B. Downey. *How to Think Like A Computer Scientist*. Green Tea Press, Needham, MA, C++ version edition, November 2012.

[71] Adam Drozdek. *Data Structures and Algorithms in C++*. Cengage Learning, Boston, MA, fourth edition, 2013.

[72] Stefanus Du Toit. Working draft, standard for programming language C++. Technical Report N3797, International Organization for Standardization and International Electrotechnical Commission, Genève, Switzerland, October 13 2013.

[73] Bruce Eckel. *Thinking in C++: Introduction to Standard C++*, volume 1. Prentice Hall, Upper Saddle River, NJ, second edition, 2000.

[74] Bruce Eckel and Chuck Allison. *Thinking in C++: Practical Programming*, volume 2. Prentice Hall, Upper Saddle River, NJ, 2003.

[75] Dirk Eddelbuettel. *Seamless R and C++ Integration with Rcpp*, volume 64 of *Use R!* Springer Science+Business Media, New York, NY, 2013.

[76] Edison Design Group staff. C++ front end: Internal documentation. Technical report, Edison Design Group, Inc., West Orange, NJ, December 21 2011.

[77] EliteHussar. Distinguish between pointers and references in C++. Available online from *cplusplus.com – The C++ Resources Network* at: http://www.cplusplus.com/articles/ENywvCM9/; October 8, 2015 was the last accessed date, August 20 2010.

[78] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. Morgan Kaufmann, San Francisco, CA, 2005.

[79] Agner Fog. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. Available online at: http://www.agner.org/optimize/optimizing_cpp.pdf; July 1, 2015 was the last accessed date, August 7 2014.

[80] Neal Ford. *The Productive Programmer: Theory in Practice*. O'Reilly Media, Sebastopol, CA, 2008.

[81] Tony Gaddis. *Starting Out With C++: From Control Structures Through Objects*. Pearson Education, Boston, MA, sixth (brief) edition, 2010.

[82] Tony Gaddis. *Starting Out With C++: From Control Structures Through Objects*. Addison-Wesley, Boston, MA, seventh edition, 2012.

[83] Tony Gaddis, Judy Walters, and Godfrey Muganda. *Starting Out With C++: Early Objects*. Addison-Wesley, Boston, MA, seventh edition, 2011.

[84] Anton Gogolev. answer to the question 'why can templates only be implemented in the header file?'. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/a/495032/1531728; edited by Jonathan Henly, September 19, 2013; March 30, 2016 was the last accessed date, January 30 2009.

[85] Sally A. Goldman and Kenneth J. Goldman. *A Practical Guide to Data Structures and Algorithms using Java*. Chapman & Hall/CRC, Boca Raton, FL, 2008.

[86] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python*. John Wiley & Sons, Hoboken, NJ, 2013.

[87] Michael T. Goodrich, Roberto Tamassia, and David M. Mount. *Data Structures and Algorithms in C++*. John Wiley & Sons, Hoboken, NJ, second edition, 2011.

[88] Peter Gottschling. *Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers*. The C++ In-Depth Series. Addison-Wesley, Boston, MA, 2016.

[89] Brad Green. Programming problems: A primer for the technical interview – fundamentals in c++11. Self-published, 2012.

[90] Marc Gregoire. *Professional C++*. John Wiley & Sons, Indianapolis, IN, third edition, 2014.

[91] Marc Gregoire, Nicholas A. Solter, and Scott J. Kleper. *Professional C++*. John Wiley & Sons, Indianapolis, IN, second edition, 2011.

[92] Richard Groff. *C++ Tutorial*. Wiffletree World, LLC, Arlington, TX, 2012.

[93] Jonathan L. Gross and Jay Yellen. *Handbook of Graph Theory*. Discrete Mathematics and Its Applications. CRC Press, Boca Raton, FL, 2004.

[94] Jonathan L. Gross, Jay Yellen, and Ping Zhang. *Handbook of Graph Theory*. Discrete Mathematics and Its Applications. CRC Press, Boca Raton, FL, 2014.

[95] Kurt Guntheroth. *Optimized C++: Proven Techniques for Heightened Performance*. O'Reilly Media, Sebastopol, CA, 2016.

[96] P. L. Hammer, E. L. Johnson, and B. H. Korte. Conclusive remarks. In *Discrete Optimization II, Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium*, volume 5 of *Annals of Discrete Mathematics*, pages 427–453. North-Holland, Banff, Alberta, Canada and Vancouver, British Columbia, Canada, August 1979.

[97] Steve Heller. *C++: A Dialog: Programming with the C++ Standard Library*. Pearson Education, Upper Saddle River, NJ, 2003.

[98] Hewlett-Packard Company staff. Standard template library programmer's guide. Available online in *SGI – The Trusted Leader in High Performance Computing: Tech Archive: Standard xTemplate Library Programmer's Guide* at: http://www.sgi.com/tech/stl/; September 30, 2015 was the last accessed date, 1994.

[99] Hewlett-Packard Company staff. STL complexity specifications. Available online in *SGI – The Trusted Leader in High Performance Computing: Tech Archive: Standard Template Library Programmer's Guide: Design documents: STL Complexity Specifications* at: http://www.sgi.com/tech/stl/complexity.html; September 30, 2015 was the last accessed date, http://www.sgi.com/tech/stl/complexity.html 2014.

[100] Leslie Hogben. *Handbook of Linear Algebra*. Discrete Mathematics and Its Applications. Chapman & Hall/CRC, Boca Raton, FL, 2007.

[101] Cay S. Horstmann. *C++ for Everyone*. John Wiley & Sons, Hoboken, NJ, second edition, 2012.

[102] Ivor Horton. *Beginning C++*. Apress, New York, NY, 2014.

[103] Justin Husted. Fixing some common compiler problems. Available online from the *UW ACM's (University of Washington's chapter of the Association for Computing Machinery) web page: Tutorials: [Development] in UNIX* at: http://www.cs.washington.edu/acm/tutorials/dev-in-unix/compiler-problems.html; February 17, 2016 was the last accessed date, May 19 2000.

[104] Innovation 24 staff. *LocalSolver 5.5 documentation: Overview*. Innovation 24, Paris, France, 2015.

[105] Innovation 24 staff. Localsolver: Mathematical optimization solver. Available online at: http://www.localsolver.com/; December 7, 2015 was the last accessed date, 2015.

[106] ISO/IEC JTC1/SC22/WG21 members. Information technology - programming languages - C++. Technical Report INCITS/ISO/IEC 14882-2011[2012], American National Standards Institute, New York, NY, February 14 2012.

[107] ISO/IEC JTC1/SC22/WG21 members. Information technology - programming languages - C++. Technical Report CAN/CSA-ISO/IEC 14882:16, Canadian Standards Association, Toronto, Canada, December 2016.

[108] ISO/IEC JTC1/SC22/WG21 members. Programming languages - C++. Technical Report ISO/IEC STANDARD 14882, International Organization for Standardization and International Electrotechnical Commission, Vernier, Genève, Switzerland, December 2017.

[109] Ted Jensen. A tutorial on pointers and arrays in C. Available online as Version 1.2 at: http://pweb.netcom.com/~tjensen/ptr/cpoint.htm, http://pweb.netcom.com/~tjensen/ptr/pointers.htm, and http://home.earthlink.net/~momotuk/pointers.pdf; self-published; October 8, 2015 was the last accessed date, September 2003.

[110] Jeffrey Johnson. *Hypernetworks in the Science of Complex Systems*, volume 3 of *Series on Complexity Science*. Imperial College Press, London, U.K., 2013.

[111] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, Reading, MA, 1999.

[112] Nicolai M. Josuttis. Josuttis' summary of STL algorithms. Available online at: http://www.josuttis.com/libbook/algolist.pdf; self-published; July 1, 2015 was the last accessed date, 1999.

[113] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Pearson Education, Upper Saddle River, NJ, second edition, 2012.

[114] Björn Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost*. Pearson Education, Boston, MA, 2006.

[115] George Em Karniadakis and Robert M. Kirby II. *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation*. Cambridge University Press, Cambridge, U.K., 2003.

[116] Jayantha Katupitiya and Kim Bentley. *Interfacing with C++: Programming Real-World Applications*. Springer-Verlag Berlin Heidelberg, Heidelberg, Germany, 2006.

[117] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, 1999.

[118] Darko Kirovski and Miodrag Potkonjak. A quantitative approach to functional debugging. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD 1997)*, pages 170–173, San Jose, CA, November 9–13 1997. IEEE Computer Society, IEEE Press.

[119] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press and Addison-Wesley, New York, NY and Reading, MA, 1993.

[120] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, third edition, 1997.

[121] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010: Mixed integer programming library version 5. *Mathematical Programming Computation*, 3(2):103–163, June 2011.

[122] Andrew Koenig and Barbara Moo. *Accelerated C++: Practical Programming by Example*. The C++ In-Depth Series. Addison-Wesley, Boston, MA, 2000.

[123] Gayle Laakmann. Cracking the technical interview: 150 technical interview questions and solutions, written by experts. Self-published, 2009.

[124] Jeff Langr. *Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better*. The Pragmatic Programmers, Raleigh, NC, October 2013.

[125] Christophe Lecoutre. *Constraint Networks: Targeting Simplicity for Techniques and Algorithms*. Wiley-ISTE. ISTE Ltd, London, U.K., 2009.

[126] Insup Lee, Joseph Y-T. Leung, and Sang H. Son. *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC Computer & Information Science. Chapman & Hall/CRC, Boca Raton, FL, 2008.

[127] Jon Lee. *A First Course in Combinatorial Optimization*, volume 36 of *Cambridge Texts in Applied Mathematics*. Cambridge University Press, New York, NY, 2004.

[128] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer*. Addison-Wesley, Upper Saddle River, NJ, fifth edition, 2013.

[129] Ray Lischner. *Exploring C++ 11: Problems and Solutions Handbook*. Apress Media, LLC, Berkeley, CA, 2013.

[130] László Lovász. *Large Networks and Graph Limits*, volume 60 of *Colloquium Publications*. American Mathematical Society, Providence, RI, 2012.

[131] László Lovász. *Geometric Representations of Graphs*. Eötvös Loránd University, Budapest, Hungary, October 17 2014.

[132] Mi Lu. *Arithmetic and Logic in Computer Systems*. John Wiley & Sons, Hoboken, NJ, 2004.

[133] Michael Main and Walter Savitch. *Data Structures and Other Objects Using C++*. Addison-Wesley, Reading, MA, fourth edition, 2011. Lecture slides based on this book are available online at: http://www.cs.colorado.edu/~main/supplements/lectures.html; September 20, 2010 was the last accessed date.

[134] D. S. Malik. *C++ Programming: From Problem Analysis to Program Design*. Course Technology, Boston, MA, fifth edition, 2011.

[135] D. S. Malik. *C++ Programming: Program Design Including Data Structures*. Course Technology, Boston, MA, fifth edition, 2011.

[136] D. S. Malik. *C++ Programming: From Problem Analysis to Program Design*. Cengage Learning, Boston, MA, sixth edition, 2013.

[137] D. S. Malik. *C++ Programming: Program Design Including Data Structures*. Cengage Learning, Boston, MA, sixth edition, 2013.

[138] D. S. Malik. *C++ Programming: From Problem Analysis to Program Design*. Cengage Learning, Stamford, CT, seventh edition, 2015.

[139] D. S. Malik. *C++ Programming: Program Design Including Data Structures*. Cengage Learning, Stamford, CT, seventh edition, 2015.

[140] D. S. Malik. *C++ Programming: From Problem Analysis to Program Design*. Cengage Learning, Boston, MA, eighth edition, 2018.

[141] Norman Matloff. *Parallel Computing for Data Science: With Examples in R, C++ and CUDA*. Chapman & Hall/CRC The R Series. CRC Press, Boca Raton, FL, 2016.

[142] Michael B. McLaughlin. *C++ Succinctly*. Succinctly. Syncfusion Inc., Morrisville, NC, 2012.

[143] Kurt McMahon. C strings and C++ strings. Available online from *Kurt McMahon's web page: Notes* at: https://www.prismnet.com/~mcmahon/Notes/strings.html; self-published; November 3, 2015 was the last accessed date.

[144] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Professional Computing Series. Pearson Education, Upper Saddle River, NJ, third edition, 2005.

[145] Scott Meyers. *Effective C++ Digital Collection: 140 Ways to Improve Your Programming*. Addison-Wesley Professional Computing Series. Addison-Wesley, Upper Saddle River, NJ, 2012.

[146] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Sebastopol, CA, 2015.

[147] Mike Mintz and Robert Ekendahl. *Hardware Verification with C++: A Practitioner's Handbook*. Springer Science+Business Media, LCC, New York, NY, 2006.

[148] Sandipan Mohanty. Programming in C++: Introduction to the current language standard C++14. Available online from *Forschungszentrum Jülich GmbH: Institute for Advanced Simulation: Jülich Supercomputing Center* at: https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/cplusplus/cplusplus.pdf and https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/cplusplus/cplusplus.pdf?__blob=publicationFile; October 3, 2017 was the last accessed date, May 30 - June 2 2016.

[149] Sandipan Mohanty. Programming in C++: Introduction to the language standard C++14 (and a preview of C++17). Available online from *Forschungszentrum Jülich GmbH: Institute for Advanced Simulation: Jülich Supercomputing Center* at: https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/cplusplus/cplusplus.pdf and https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/cplusplus/cplusplus.pdf;jsessionid=F6F2721BAADDF0E098B19D576F1E0ED1?__blob=publicationFile; self-published; June 22, 2018 was the last accessed date, May 29 – June 1 2017.

[150] Mohtashim. C++ STL tutorial. Available online at *Tutorials Point: C++ Tutorial: C++ STL Tutorial*: http://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm; September 17, 2015 was the last accessed date, 2015.

[151] Peter Mortensen. What is the difference between const int*, const int * const, and int const *? Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/questions/1143262/what-is-the-difference-between-const-int-const-int-const-and-int-const; October 1, 2015 was the last accessed date, March 13 2015.

[152] Black Moses. answer to the question 'why is the discrepancy in these two cases of using C++ templates?'. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/a/36319466/1531728 and http://stackoverflow.com/questions/36319028/why-is-the-discrepancy-in-these-two-cases-of-using-c-templates/36319466#36319466; edited by Kuba Ober, on March 30, 2016; March 30, 2016 was the last accessed date, March 30 2016.

[153] Arindam Mukherjee. *Learning Boost C++ Libraries: Solve Practical Programming Problems Using Powerful, Portable, and Expressive Libraries from Boost*. Packt Publishing, Birmingham, West Midlands, England, U.K., July 2015.

[154] Carlos Oliveira. *Practical C++ Financial Programming: Practical Solutions in Financial Engineering*. The Expert's Voice$^{\circ}$ in Programming. Apress Media, LLC, Berkeley, CA, 2015.

[155] Mikael Olsson. *C++ Quick Syntax Reference*. The Expert's Voice$^{\circ}$. Apress Media, LLC, New York, NY, 2013.

[156] Steve Oualline. *Practical C++ Programming*. Programming Style Guidelines. O'Reilly Media, Sebastopol, CA, second edition, 2003.

[157] Brian Overland. *C++ Without Fear: A Beginner's Guide That Makes You Feel Smart*. Pearson Education, Boston, MA, second edition, 2011.

[158] Brian Overland. *C++ for the Impatient*. Pearson Education, Upper Saddle River, NJ, 2013.

[159] Yusuf Kemal Özcan. Is there any difference between pointers and references? Available online from *Stack Exchange Inc.: Programmers Stack Exchange: Questions* at: http://programmers.stackexchange.com/questions/195337/is-there-any-difference-between-pointers-and-references; October 28, 2015 was the last accessed date, April 18 2013.

[160] Alonso Peña. *Advanced Quantitative Finance with C++*. Community Experience Distilled. Packt Publishing, Birmingham, West Midlands, England, U.K., 2014.

[161] Rod Pierce. Maths is fun: Maths resources. Available online at: https://www.mathsisfun.com/; February 5, 2016 was the last accessed date, 2016.

[162] Joe Pitt-Francis and Jonathan Whiteley. *Guide to Scientific Computing in C++*. Undergraduate Topics in Computer Science. Springer-Verlag London, London, U.K., 2012.

[163] Antony Polukhin. *Boost C++ Application Development Cookbook: Over 80 practical, task-based recipes to create applications using Boost libraries*. Packt Publishing, Birmingham, West Midlands, England, U.K., 2013.

[164] Alex Pomeranz. Learn C++. Available online at: http://www.learncpp.com/; April 2, 2014 was the last accessed date, 2014.

[165] Constantine Pozrikidis. *Introduction to C++ Programming and Graphics*. Springer Science+Business Media, LCC, New York, NY, 2007.

[166] Stephen Prata. *C++ Primer Plus*. Sams Publishing, Indianapolis, IN, fifth edition, 2005.

[167] Stephen Prata. *C++ Primer Plus: Developer's Library*. Pearson Education, Upper Saddle River, NJ, sixth edition, 2012.

[168] Programming Research Ltd. staff. High integrity C++: Coding standard version 4.0. Technical report, Programming Research Ltd., Hersham, Surrey, England, U.K., October 3 2013.

[169] Siddhartha Rao. *Sams Teach Yourself C++ in One Hour a Day*. Sams Publishing, Indianapolis, IN, eighth edition, 2017.

[170] Greg Reese. *C++ Standard Library Practical Tips*. Charles River Media Programming Series. Charles River Media, Hingham, MA, 2006.

[171] Chris Riesbeck. Standard C++ containers. Available online from *Prof. Chris Riesbeck's web page: Programming: Useful C++ / Unix Resources*, Computer Science Division, Department of Electrical Engineering and Computer Science, Robert R. McCormick School of Engineering and Applied Science, Northwestern University at: http://www.cs.northwestern.edu/~riesbeck/programming/c++/stl-summary.html; September 30, 2015 was the last accessed date, July 3 2009.

[172] Chris Riesbeck. Useful C++ / Unix resources. Available online from *Prof. Chris Riesbeck's web page: Programming*, Computer Science Division, Department of Electrical Engineering and Computer Science, Robert R. McCormick School of Engineering and Applied Science, Northwestern University at: http://www.cs.northwestern.edu/~riesbeck/programming/c++/; September 30, 2015 was the last accessed date, July 2 2009.

[173] Robert Robson. *Using the STL: The C++ Standard Template Library*. Springer-Verlag Berlin Heidelberg New York, Heidelberg, Germany, second edition, 2000.

[174] Philip Romanik and Amy Muntz. *Applied C++: Practical Techniques for Building Better Software*. The C++ In-Depth Series. Addison-Wesley, Boston, MA, 2003.

[175] Stephan Roth. *Clean C++: Sustainable Software Development Patterns and Best Practices with C++ 17*. Apress Media, LLC, Berkeley, CA, 2017.

[176] Dan Saks. An introduction to references. Available online from *UBM Electronics: UBM Canon Electronics Engineering Communities: Embedded – Cracking the Code to Systems Development* at: http://www.embedded.com/print/4024641; October 8, 2015 was the last accessed date, February 26 2001.

[177] Dan Saks. References vs. pointers. Available online from *UBM Electronics: UBM Canon Electronics Engineering Communities: Embedded – Cracking the Code to Systems Development* at: http://www.embedded.com/electronics-blogs/programming-pointers/4023307/References-vs-Pointers and http://www.embedded.com/print/4023307; October 28, 2015 was the last accessed date, March 15 2001.

[178] Majid Sarrafzadeh and C. K. Wong. *An Introduction to VLSI Physical Design*. McGraw-Hill Series in Computer Science. McGraw-Hill, New York, NY, 1996.

[179] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, Reading, MA, 1998.

[180] Walter Savitch. *Problem Solving with C++*. Pearson Education, Boston, MA, seventh edition, 2009.

[181] Walter Savitch. *Problem Solving with C++*. Addison-Wesley, Reading, MA, eighth edition, 2012.

[182] Walter Savitch and Kenrick Mock. *Absolute C++*. Addison-Wesley, Reading, MA, fifth edition, 2013.

[183] Boris Schäling. The boost C++ libraries. Available online at: http://en.highscore.de/cpp/boost/; self-published; February 26, 2013 was the last accessed date, 2012.

[184] Edward Scheinerman. *C++ for Mathematicians: An Introduction for Students and Professionals*. Chapman & Hall/CRC, Boca Raton, FL, 2006.

[185] Edward R. Scheinerman and Daniel H. Ullman. *Fractional Graph Theory: A Rational Approach to the Theory of Graphs*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, New York, NY, 1997.

[186] Herbert Schildt. *C++: The Complete Reference*. McGraw-Hill, Berkeley, CA, third edition, 1998.

[187] Herbert Schildt. *C++ from the Ground Up*. McGraw-Hill/Osborne, Berkeley, CA, third edition, 2003.

[188] Herbert Schildt. *C++: The Complete Reference*. Osborne Complete Reference Series. McGraw-Hill/Osborne, Berkeley, CA, fourth edition, 2003.

[189] Herbert Schildt. *The Art of C++*. McGraw-Hill/Osborne, Emeryville, CA, 2004.

[190] sdsmith. comment to the question 'why is the discrepancy in these two cases of using C++ templates?'. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/q/36319028/1531728; March 30, 2016 was the last accessed date, March 30 2016.

[191] Robert C. Seacord. *Secure Coding in C and C++*. The SEI Series in Software Engineering. Addison-Wesley, Upper Saddle River, NJ, second edition, 2013.

[192] Yair Shapira. *Solving PDEs in C++: Numerical Methods in a Unified Object-Oriented Approach*. SIAM Computational Science and Engineering. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2006.

[193] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag London, London, U.K., second edition, 2008.

[194] Richard Smith. Working draft, standard for programming language C++. Technical Report N4659, International Organization for Standardization and International Electrotechnical Commission, Genève, Switzerland, March 21 2017.

[195] Juan Soulié. *C++ Language Tutorial*. cplusplus.com, June 2007.

[196] Daniel A. Spielman. Spectral graph theory and its applications. In *Proceedings of the* 48^{th} *IEEE Symposium on Foundations of Computer Science (FOCS '07)*, pages 29–38, Providence, RI, October 21–23 2007. IEEE Computer Society Press.

[197] Milan Stevanovic. *Advanced C and C++ Compiling: An engineering guide to compiling, linking, and libraries using C and C++*. Apress, New York, NY, 2014.

[198] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. Pearson Education, Boston, MA, 2009.

[199] Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, Upper Saddle River, NJ, fourth edition, 2013.

[200] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. Pearson Education, Upper Saddle River, NJ, second edition, 2014.

[201] Bjarne Stroustrup. *A Tour of C++*. The C++ In-Depth Series. Pearson Education, Upper Saddle River, NJ, 2014.

[202] Bruce Sutherland. *C++ Game Development Primer*. Apress Media, LLC, Berkeley, CA, 2014.

[203] Bruce Sutherland. *Learn C++ for Game Development*. Apress Media, LLC, Berkeley, CA, 2014.

[204] Bruce Sutherland. *C++ Recipes: A Problem-Solution Approach*. Apress Media, LLC, Berkeley, CA, 2015.

[205] Bruce A. Tate. *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*. Pragmatic Bookshelf. The Pragmatic Programmers, Raleigh, NC, 2010.

[206] Mitchell Aaron Thornton, Rolf Drechsler, and D. Michael Miller. *Spectral Techniques in VLSI CAD*. Kluwer Academic Publishers, Boston, MA, 2001.

[207] Krishnaiyan "KT" Thulasiraman, Subramanian Arumugam, Andreas Brandstädt, and Takao Nishizeki. *Handbook of Graph Theory, Combinatorial Optimization, and Algorithms*. Chapman & Hall/CRC Computer & Information Science. CRC Press, Boca Raton, FL, 2016.

[208] Piet van Mieghem. *Graph Spectra for Complex Networks*. Cambridge University Press, New York, NY, 2011.

[209] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Pearson Education, Boston, MA, 2003.

[210] Dirk Vermeir. *Multi-Paradigm Programming using C++*. Springer-Verlag London Berlin Heidelberg, London, U.K., 2001.

[211] Vibrant Publishers. *C++ Internals: Interview Questions You'll Most Likely Be Asked*. Job Interview Questions Series. Vibrant Publishers, second edition, 2012.

[212] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Pearson Education, Upper Saddle River, NJ, fourth edition, 2014.

[213] Wikibooks contributors. *More C++ Idioms*. Wikimedia Foundation, San Francisco, CA, June 18 2014.

[214] Wikipedia contributors. Algebraic modeling language. Available online in *Wikipedia, The Free Encyclopedia: Specification languages* at: https://en.wikipedia.org/wiki/Algebraic_modeling_language; December 11, 2015 was the last accessed date, May 3 2015.

[215] Wikipedia contributors. Discrete optimization. Available online in *Wikipedia, The Free Encyclopedia: Mathematical optimization* at: https://en.wikipedia.org/wiki/Discrete_optimization; December 9, 2015 was the last accessed date, May 17 2015.

[216] Wikipedia contributors. Vertex separator. Available online from *Wikipedia, The Free Encyclopedia: Graph connectivity* at: https://en.wikipedia.org/wiki/Vertex_separator; November 25, 2018 was the last accessed date, November 29 2016.

[217] Wikipedia contributors. Graph labeling. Available online from *Wikipedia, The Free Encyclopedia: Extensions and generalizations of graphs* at: https://en.wikipedia.org/wiki/Graph_labeling; November 25, 2018 was the last accessed date, August 30 2017.

[218] Wikipedia contributors. Loop (graph theory). Available online from *Wikipedia, The Free Encyclopedia: Graph theory* at: https://en.wikipedia.org/wiki/Loop_(graph_theory); November 25, 2018 was the last accessed date, December 16 2017.

[219] Wikipedia contributors. Vertex (graph theory). Available online from *Wikipedia, The Free Encyclopedia: Graph theory* at: https://en.wikipedia.org/wiki/Vertex_(graph_theory); November 20, 2018 was the last accessed date, October 24 2017.

[220] Wikipedia contributors. Bouquet graph. Available online from *Wikipedia, The Free Encyclopedia: Parametric families of graphs* at: https://en.wikipedia.org/wiki/Bouquet_graph; November 25, 2018 was the last accessed date, August 2 2018.

[221] Wikipedia contributors. Degree (graph theory). Available online from *Wikipedia, The Free Encyclopedia: Graph theory* at: https://en.wikipedia.org/wiki/Degree_(graph_theory); November 20, 2018 was the last accessed date, November 12 2018.

[222] Wikipedia contributors. Depth-first search. Available online from *Wikipedia, The Free Encyclopedia: Graph algorithms* at: https://en.wikipedia.org/wiki/Depth-first_search; February 22, 2019 was the last accessed date, September 22 2018.

[223] Wikipedia contributors. Dual graph. Available online from *Wikipedia, The Free Encyclopedia: Graph operations* at: https://en.wikipedia.org/wiki/Dual_graph; November 25, 2018 was the last accessed date, May 18 2018.

[224] Wikipedia contributors. Glossary of graph theory terms. Available online from *Wikipedia, The Free Encyclopedia: Graph theory* at: https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms; November 25, 2018 was the last accessed date, November 15 2018.

[225] Wikipedia contributors. Graph (abstract data type). Available online from *Wikipedia, The Free Encyclopedia: Abstract data types* at: https://en.wikipedia.org/wiki/Graph_(abstract_data_type); November 25, 2018 was the last accessed date, September 2 2018.

[226] Wikipedia contributors. Graph (discrete mathematics). Available online from *Wikipedia, The Free Encyclopedia: Graph theory* at: https://en.wikipedia.org/wiki/Graph_(discrete_mathematics); November 25, 2018 was the last accessed date, November 19 2018.

[227] Wikipedia contributors. Handshaking lemma. Available online from *Wikipedia, The Free Encyclopedia: Graph theory* at: https://en.wikipedia.org/wiki/Handshaking_lemma; November 25, 2018 was the last accessed date, March 28 2018.

[228] Wikipedia contributors. Independent set (graph theory). Available online from *Wikipedia, The Free Encyclopedia: Computational problems in graph theory* at: https://en.wikipedia.org/wiki/Independent_set_(graph_theory); November 25, 2018 was the last accessed date, November 16 2018.

[229] Wikipedia contributors. Mixed graph. Available online from *Wikipedia, The Free Encyclopedia: Graph theory* at: https://en.wikipedia.org/wiki/Mixed_graph; February 21, 2019 was the last accessed date, May 16 2018.

[230] Wikipedia contributors. Multidimensional network. Available online from *Wikipedia, The Free Encyclopedia: Network theory* at: https://en.wikipedia.org/wiki/Multidimensional_network; November 25, 2018 was the last accessed date, November 14 2018.

[231] Wikipedia contributors. Multigraph. Available online from *Wikipedia, The Free Encyclopedia: Extensions and generalizations of graphs* at: https://en.wikipedia.org/wiki/Multigraph; November 20, 2018 was the last accessed date, May 30 2018.

[232] Wikipedia contributors. Multipartite graph. Available online from *Wikipedia, The Free Encyclopedia: Graph families* at: https://en.wikipedia.org/wiki/Multipartite_graph; November 25, 2018 was the last accessed date, May 30 2018.

[233] Wikipedia contributors. Planar graph. Available online from *Wikipedia, The Free Encyclopedia: Graph families* at: https://en.wikipedia.org/wiki/Planar_graph; November 25, 2018 was the last accessed date, November 16 2018.

[234] Wikipedia contributors. Quiver (mathematics). Available online from *Wikipedia, The Free Encyclopedia: Directed graphs* at: https://en.wikipedia.org/wiki/Quiver_(mathematics); November 25, 2018 was the last accessed date, November 10 2018.

[235] Wikipedia contributors. Breadth-first search. Available online from *Wikipedia, The Free Encyclopedia: Graph algorithms* at: https://en.wikipedia.org/wiki/Breadth-first_search; February 22, 2019 was the last accessed date, January 20 2019.

[236] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Publications Co., Shelter Island, NY, 2012.

[237] Matthew Wilson. *Extended STL: Collections and Iterators*, volume 1. Addison-Wesley, Upper Saddle River, NJ, 2007.

[238] Wolfram Research staff. Wolfram MathWorld: The web's most extensive mathematics resource. Available online at: http://mathworld.wolfram.com/; February 5, 2016 was the last accessed date, January 29 2016.

[239] Diane Zak. *An Introduction to Programming with C++*. Course Technology, Boston, MA, seventh edition, 2013.