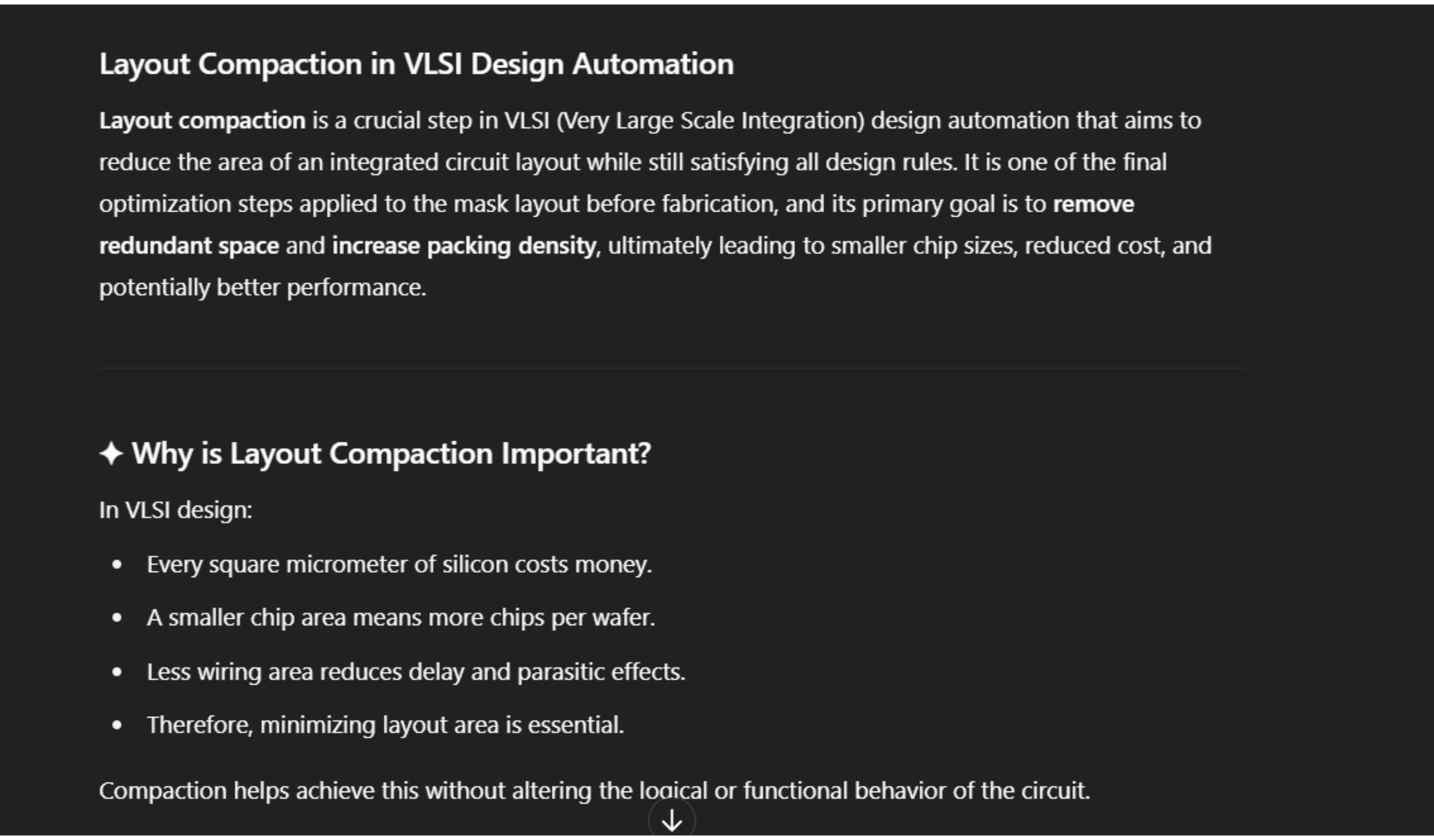


Layout Compaction in VLSI Design Automation

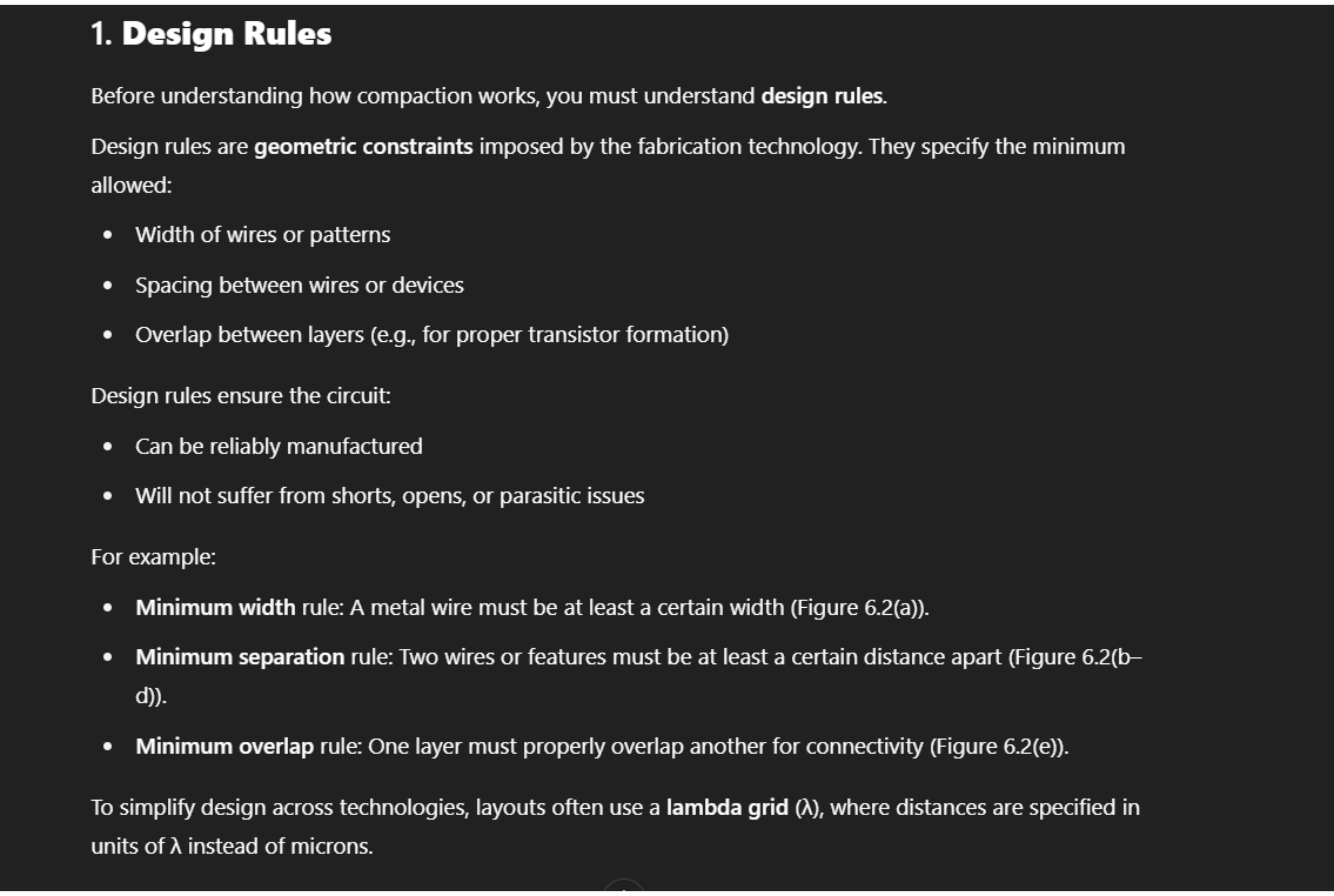
Layout compaction is a crucial step in VLSI (Very Large Scale Integration) design automation that aims to reduce the area of an integrated circuit layout while still satisfying all design rules. It is one of the final optimization steps applied to the mask layout before fabrication, and its primary goal is to **remove redundant space** and **increase packing density**, ultimately leading to smaller chip sizes, reduced cost, and potentially better performance.

✦ Why is Layout Compaction Important?

In VLSI design:

- Every square micrometer of silicon costs money.
- A smaller chip area means more chips per wafer.
- Less wiring area reduces delay and parasitic effects.
- Therefore, minimizing layout area is essential.

Compaction helps achieve this without altering the logical or functional behavior of the circuit.



1. **Design Rules**

Before understanding how compaction works, you must understand **design rules**.

Design rules are **geometric constraints** imposed by the fabrication technology. They specify the minimum allowed:

- Width of wires or patterns
- Spacing between wires or devices
- Overlap between layers (e.g., for proper transistor formation)

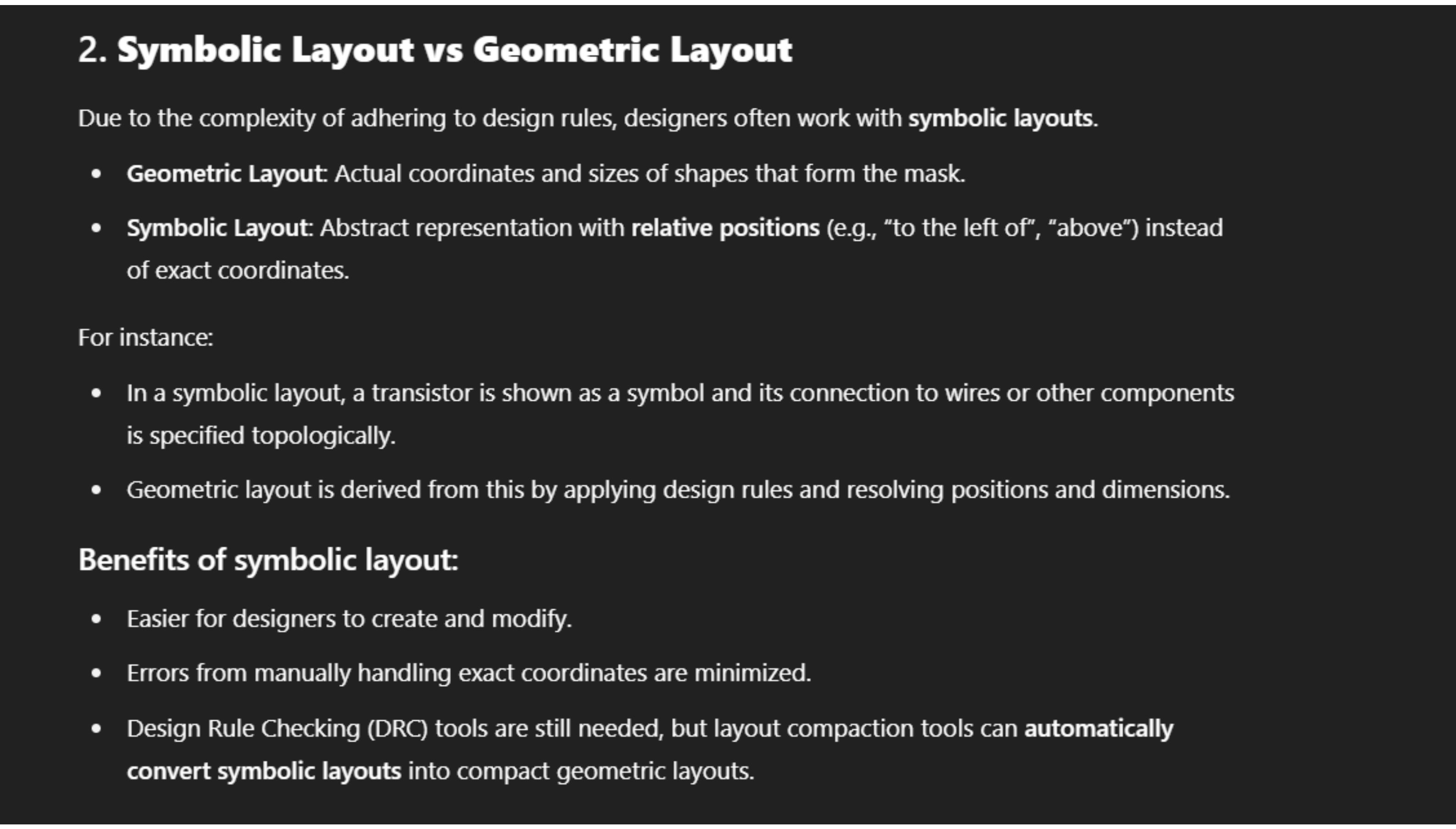
Design rules ensure the circuit:

- Can be reliably manufactured
- Will not suffer from shorts, opens, or parasitic issues

For example:

- **Minimum width** rule: A metal wire must be at least a certain width (Figure 6.2(a)).
- **Minimum separation** rule: Two wires or features must be at least a certain distance apart (Figure 6.2(b–d)).
- **Minimum overlap** rule: One layer must properly overlap another for connectivity (Figure 6.2(e)).

To simplify design across technologies, layouts often use a **lambda grid** (λ), where distances are specified in units of λ instead of microns.



2. **Symbolic Layout vs Geometric Layout**

Due to the complexity of adhering to design rules, designers often work with **symbolic layouts**.

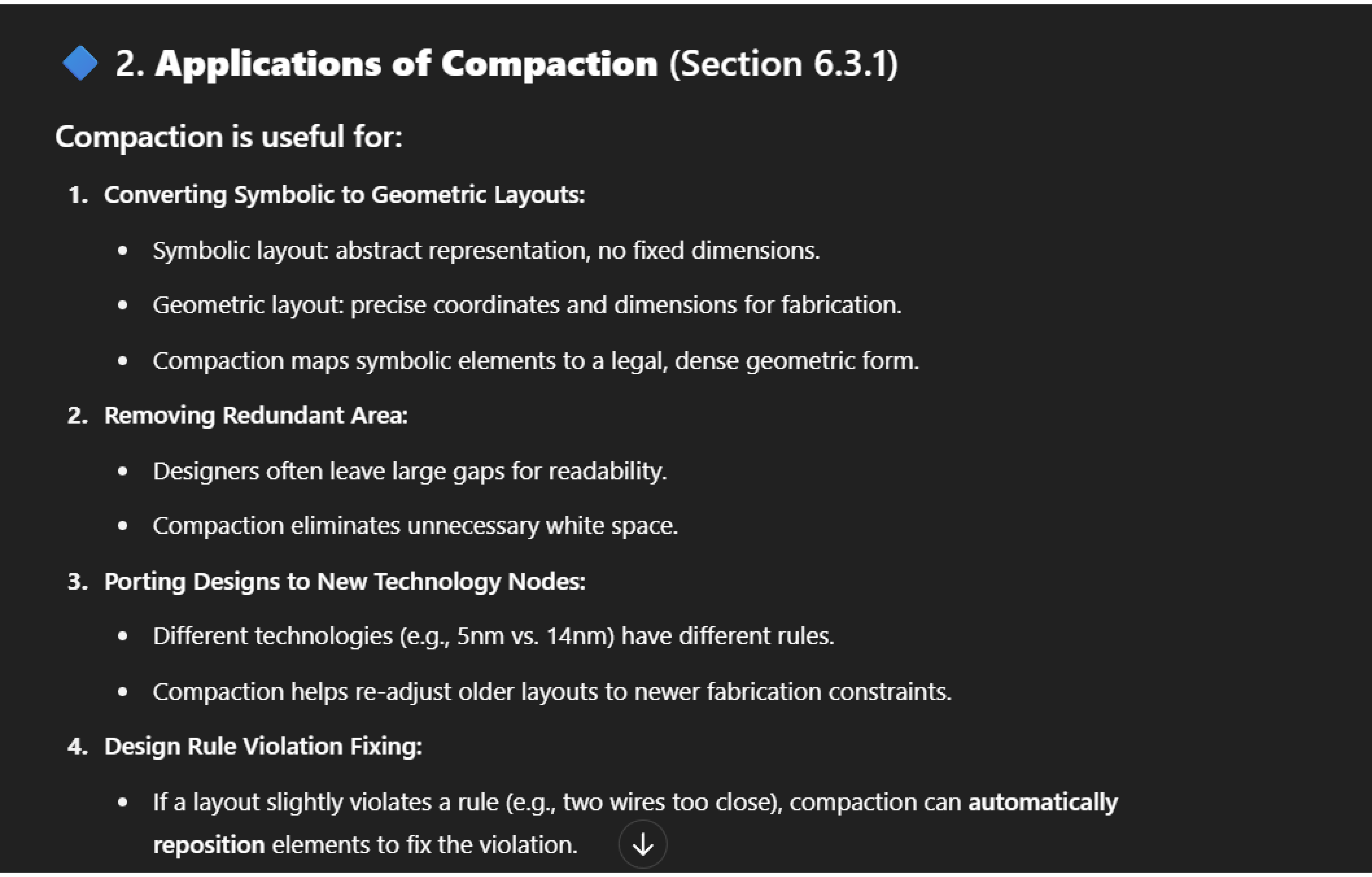
- **Geometric Layout**: Actual coordinates and sizes of shapes that form the mask.
- **Symbolic Layout**: Abstract representation with **relative positions** (e.g., "to the left of", "above") instead of exact coordinates.

For instance:

- In a symbolic layout, a transistor is shown as a symbol and its connection to wires or other components is specified topologically.
- Geometric layout is derived from this by applying design rules and resolving positions and dimensions.

Benefits of symbolic layout:

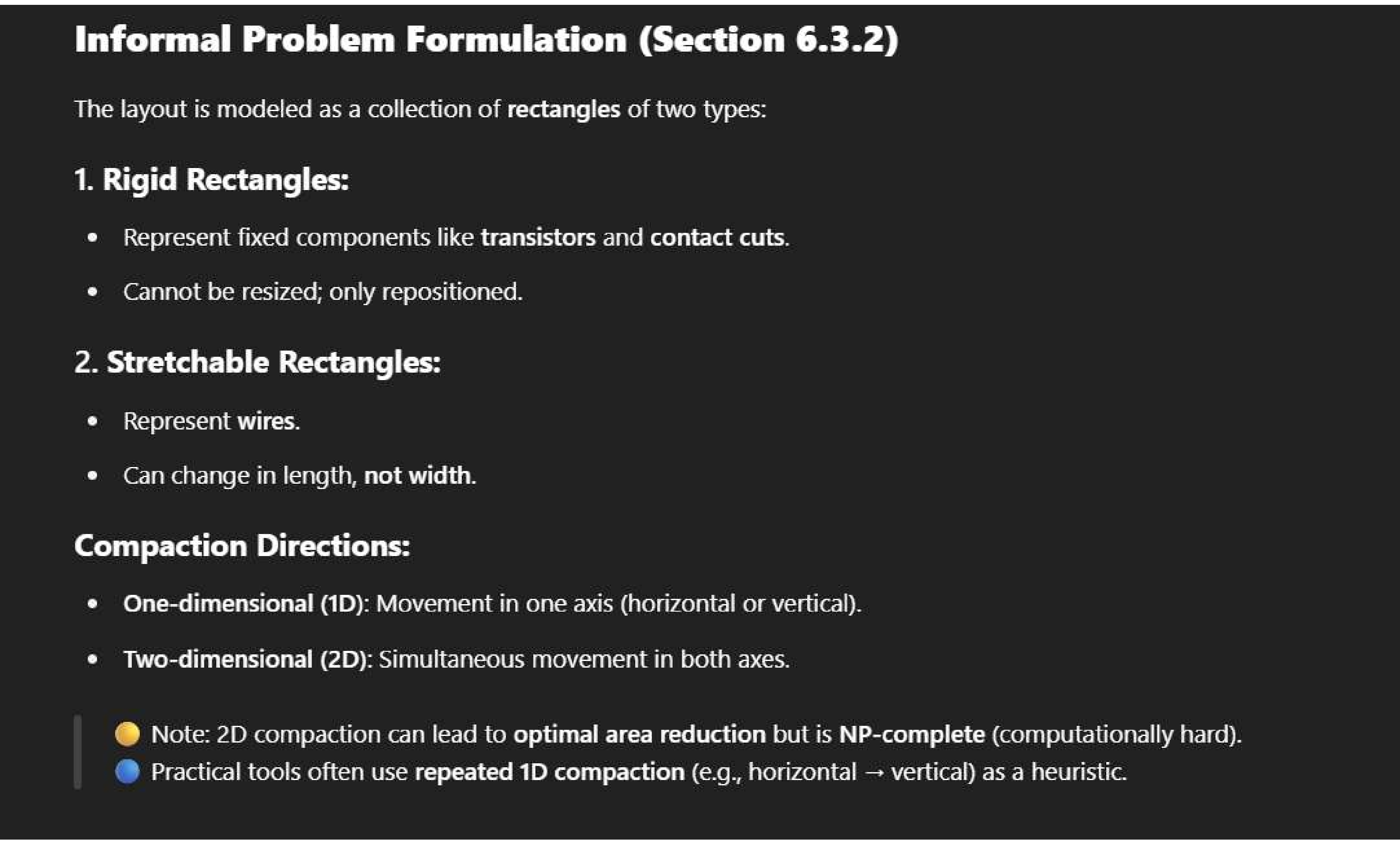
- Easier for designers to create and modify.
- Errors from manually handling exact coordinates are minimized.
- Design Rule Checking (DRC) tools are still needed, but layout compaction tools can **automatically convert symbolic layouts** into compact geometric layouts.



🔷 2. **Applications of Compaction** (Section 6.3.1)

Compaction is useful for:

1. **Converting Symbolic to Geometric Layouts:**
   - Symbolic layout: abstract representation, no fixed dimensions.
   - Geometric layout: precise coordinates and dimensions for fabrication.
   - Compaction maps symbolic elements to a legal, dense geometric form.
2. **Removing Redundant Area:**
   - Designers often leave large gaps for readability.
   - Compaction eliminates unnecessary white space.
3. **Porting Designs to New Technology Nodes:**
   - Different technologies (e.g., 5nm vs. 14nm) have different rules.
   - Compaction helps re-adjust older layouts to newer fabrication constraints.
4. **Design Rule Violation Fixing:**
   - If a layout slightly violates a rule (e.g., two wires too close), compaction can **automatically reposition** elements to fix the violation.



Informal Problem Formulation (Section 6.3.2)

The layout is modeled as a collection of **rectangles** of two types:

1. Rigid Rectangles:

- Represent fixed components like **transistors** and **contact cuts**.
- Cannot be resized; only repositioned.

2. Stretchable Rectangles:

- Represent **wires**.
- Can change in length, **not width**.

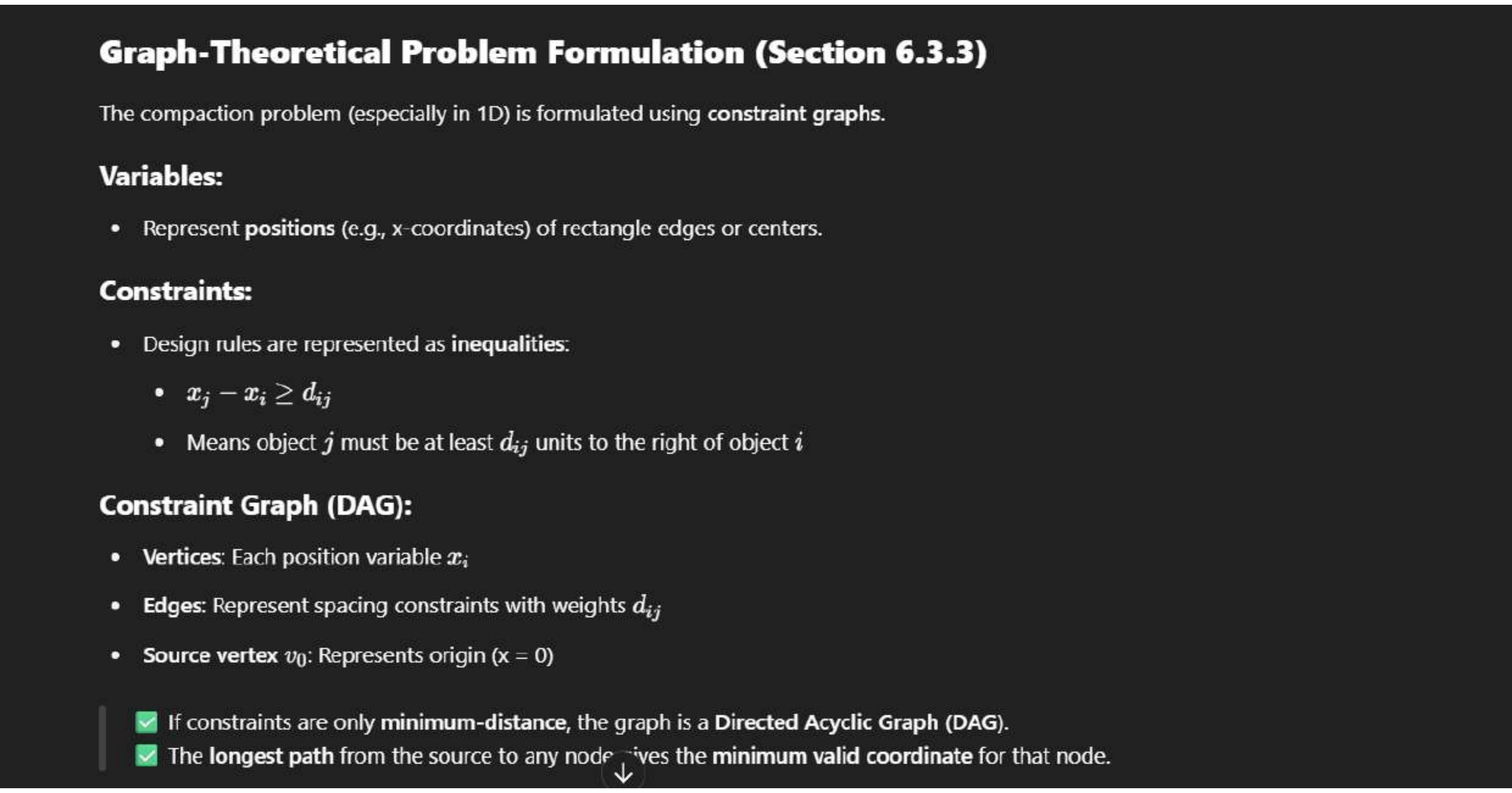
Compaction Directions:

- **One-dimensional (1D)**: Movement in one axis (horizontal or vertical).
- **Two-dimensional (2D)**: Simultaneous movement in both axes.

> 🟡 Note: 2D compaction can lead to **optimal area reduction** but is **NP-complete** (computationally hard).
> 🔵 Practical tools often use **repeated 1D compaction** (e.g., horizontal → vertical) as a heuristic.

Problem Example:

- A layout with redundant spacing and 9 rigid rectangles.
- Different compaction orders (horizontal first or vertical first) yield **different results.**
- Only a **true 2D compaction** achieves the optimal layout (minimum area).
- Shows why 1D compaction, although useful, can fall short.



Graph-Theoretical Problem Formulation (Section 6.3.3)

The compaction problem (especially in 1D) is formulated using **constraint graphs**.

Variables:

- Represent **positions** (e.g., x-coordinates) of rectangle edges or centers.

Constraints:

- Design rules are represented as **inequalities**:
  - $x_j - x_i \geq d_{ij}$
  - Means object j must be at least d_{ij} units to the right of object i

Constraint Graph (DAG):

- **Vertices**: Each position variable x_i
- **Edges**: Represent spacing constraints with weights d_{ij}
- **Source vertex** v_0: Represents origin (x = 0)

> ✅ If constraints are only **minimum-distance**, the graph is a **Directed Acyclic Graph (DAG)**.
> ✅ The **longest path** from the source to any node gives the **minimum valid coordinate** for that node.

Extended Problem with Maximum-Distance Constraints (Section 6.3.4)

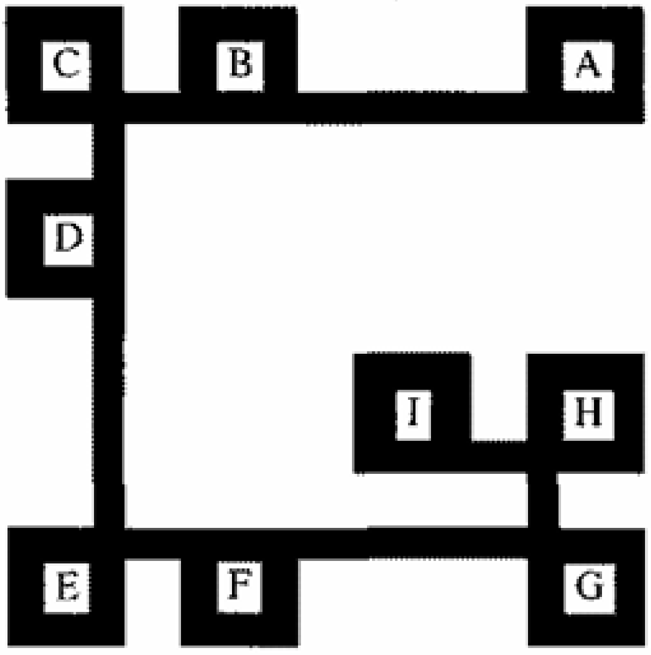
Sometimes, there are **maximum distance** constraints (e.g., to preserve electrical connectivity).

Example:

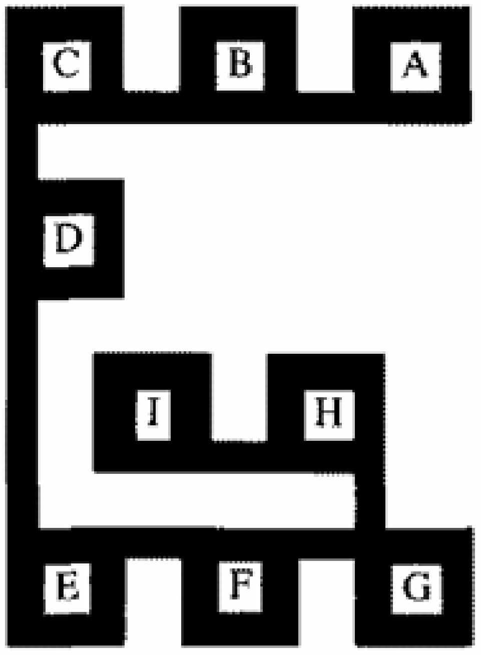
- A wire cannot be longer than d, so:
  - $x_c - x_w \leq d$
  - Which can be rewritten as:
    - $x_w - x_c \geq -d$

This leads to:

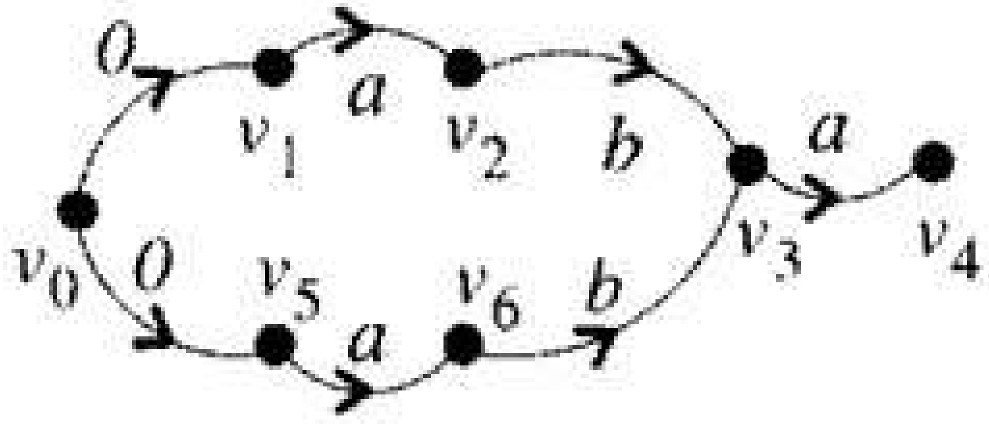
- **Negative-weight edges** in the constraint graph.
- May introduce **cycles**.
- Solving becomes harder: no longer just longest paths in DAGs.



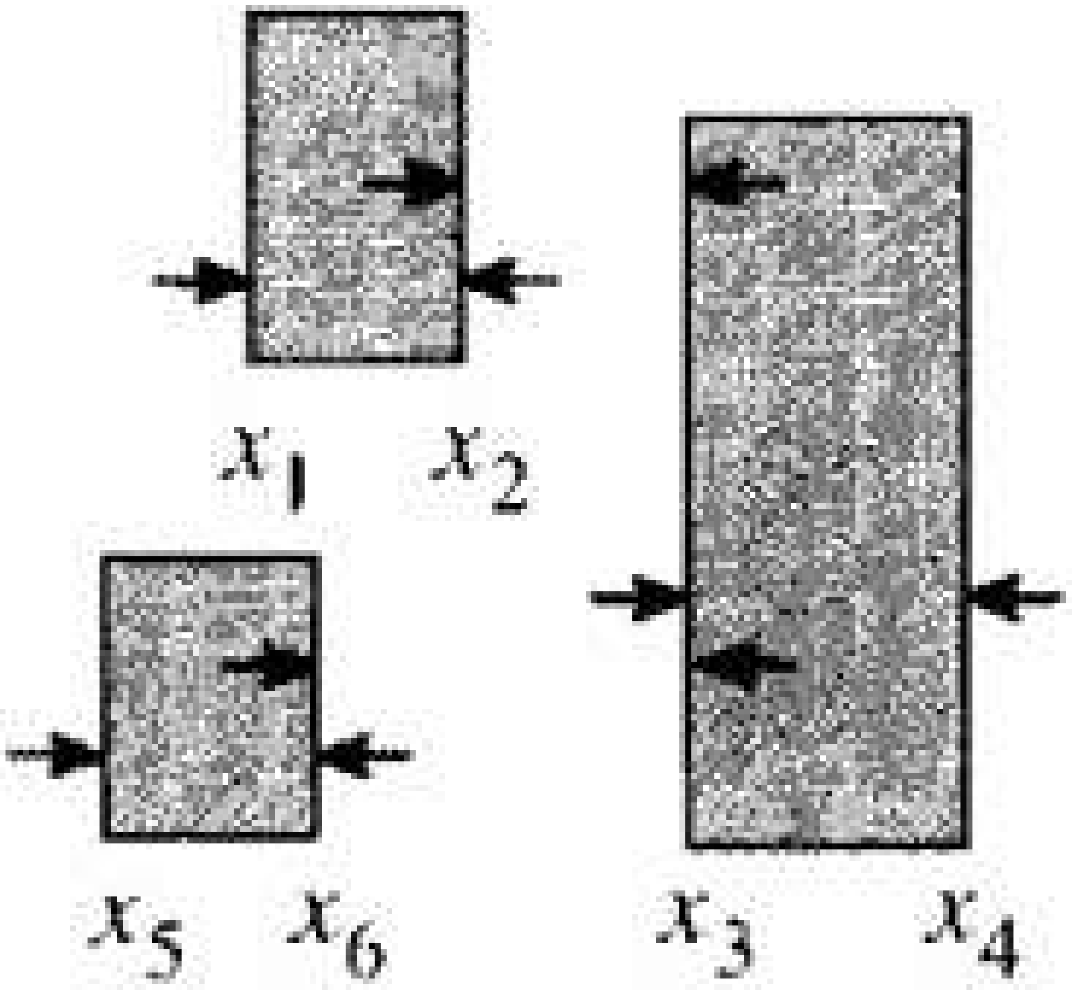
(a)



(b)



0
a
b
a
0
a
b
v_0
v_1
v_2
v_3
v_4
v_5
v_6


x_1
x_2
x_5
x_6
x_3
x_4

Longest path in a directed Acyclic graph | Dynamic Programming

Last Updated : 30 Dec, 2021

Given a directed graph **G** with *N vertices* and *M edges*. The task is to find the length of the longest directed path in Graph.

ou are given a Weighted Directed Acyclic Graph (DAG) consisting of ‘N’ nodes and ‘E’ directed edges. Nodes are numbered from 0 to ’N’-1. You are also given a source node ‘Src’ in t. Your task is to find the longest distances from ‘Src’ to all the nodes in the given graph.

Return an array of ‘N’ integers where ‘ith’ integer gives the maximum distance of the node ‘i’ rom the source node ‘Src’.

A Directed Acyclic Graph (DAG) is a directed graph that contains no cycles.

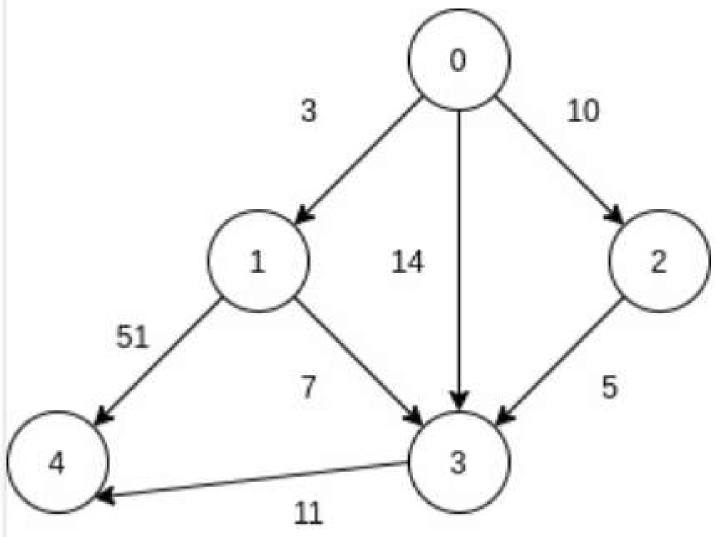
Note:

Print -1 if a node is not reachable from the given source node.

Print -1 if a node is not reachable from the given source node.

Example:

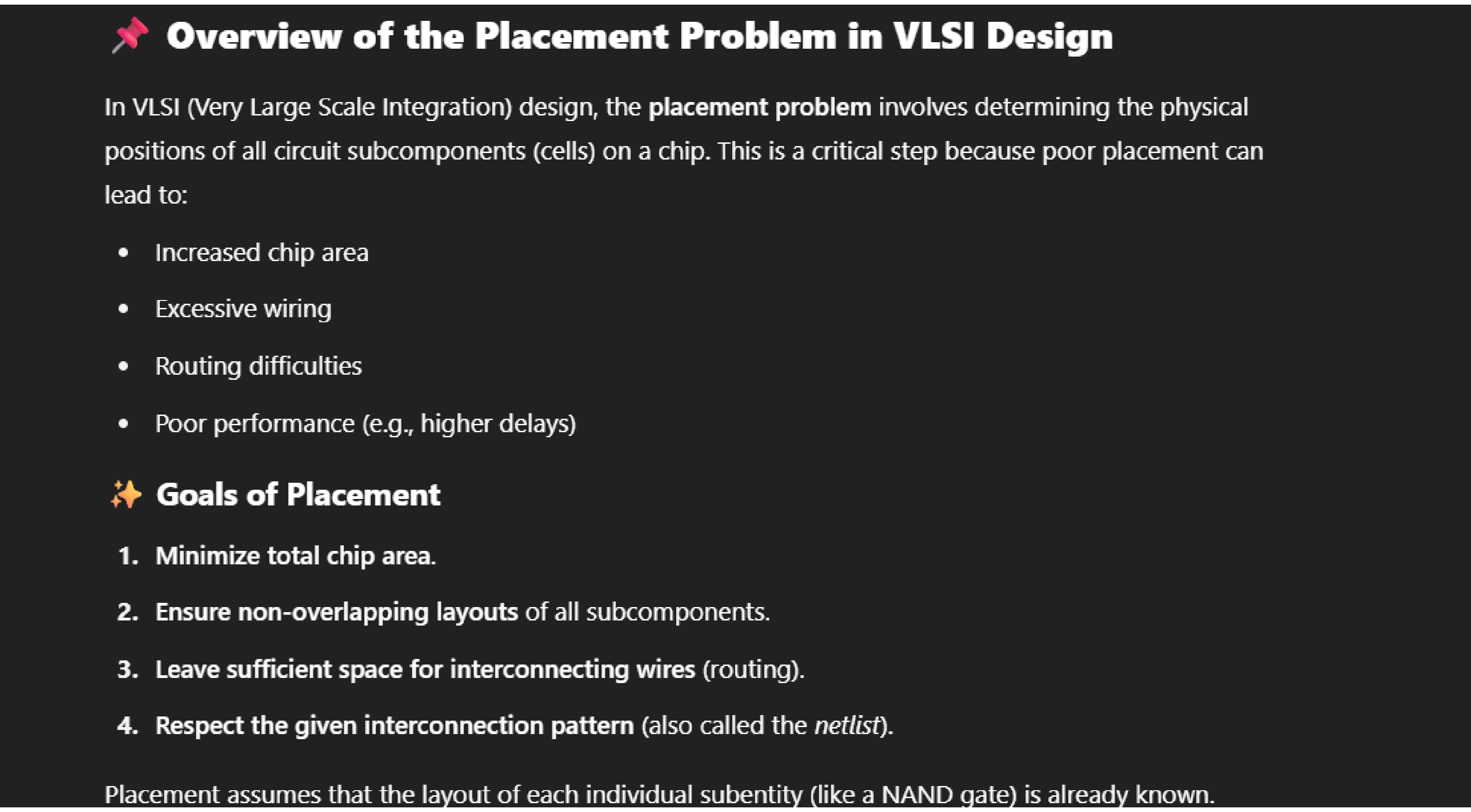
Consider the following DAG consists of 5 nodes and 7 edges, Let the source node ‘Src’ be 0.



Then the maximum distance of node 0 from the source node 0 is 0. (the distance of a node from itself is always 0).
The maximum distance of node 1 from the source node 0 is 3. The path that gives this maximum distance is 0 -> 1.
The maximum distance of node 2 from the source node 0 is 10. The path that gives this maximum distance is 0 -> 2.
The maximum distance of node 3 from the source node 0 is 15. The path that gives this maximum distance is 0 -> 2 -> 3.
The maximum distance of node 4 from the source node 0 is 54. The path that gives this maximum distance is 0 -> 1 -> 4.

Thus we should print 0 3 10 15 54

Simple Approach: A naive approach is to calculate the length of the longest path from every node using DFS.
The time complexity of this approach is O(N^2).
Efficient Approach: An efficient approach is to use Dynamic Programming and DFS together to find the longest path in the Graph.
Let **dp[i]** be the length of the longest path starting from the node **i**. Initially all positions of dp will be 0. We can call the DFS function from every node and traverse for all its children. The recursive formula will be:



📌 Overview of the Placement Problem in VLSI Design

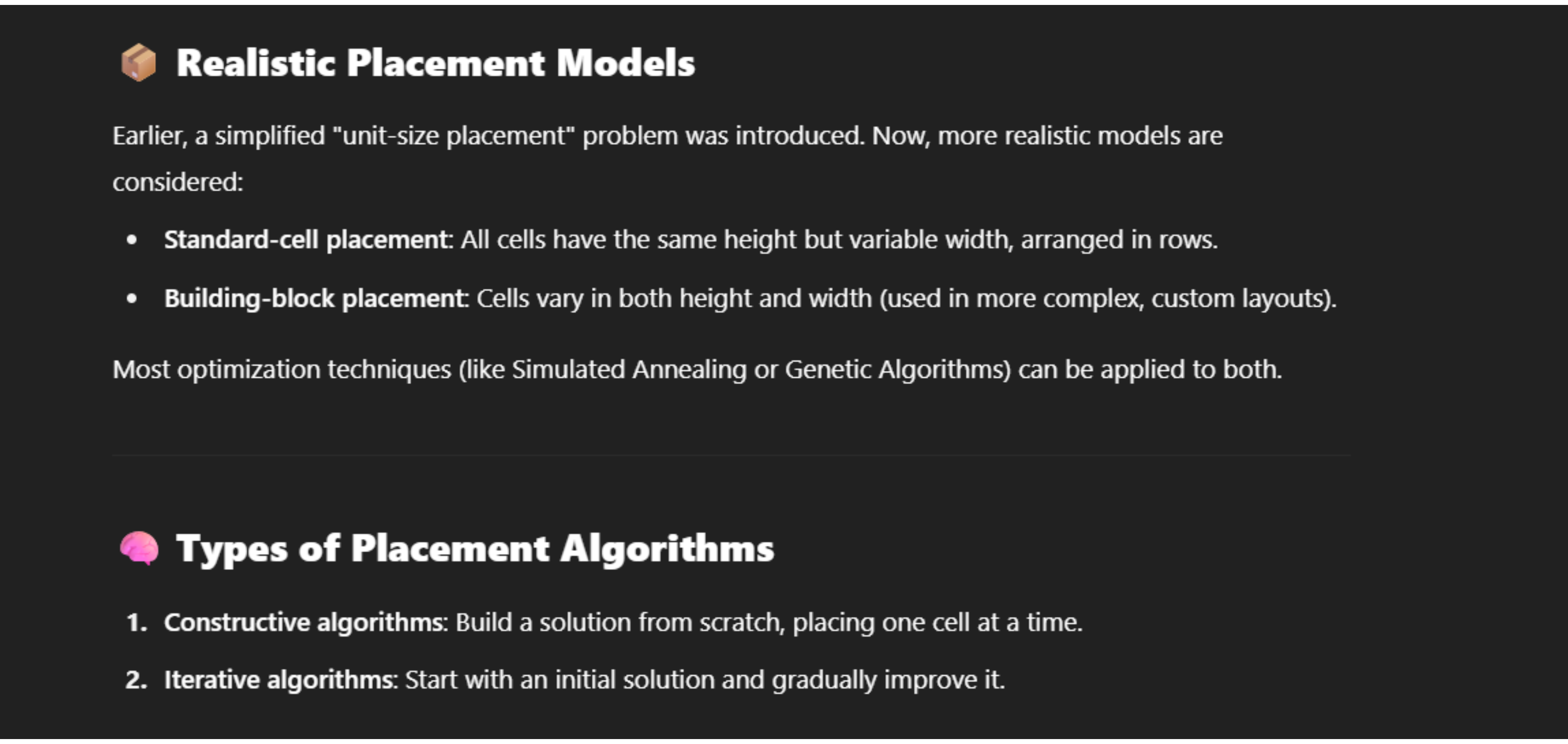
In VLSI (Very Large Scale Integration) design, the **placement problem** involves determining the physical positions of all circuit subcomponents (cells) on a chip. This is a critical step because poor placement can lead to:

- Increased chip area
- Excessive wiring
- Routing difficulties
- Poor performance (e.g., higher delays)

✨ Goals of Placement

1. **Minimize total chip area.**
2. **Ensure non-overlapping layouts** of all subcomponents.
3. **Leave sufficient space for interconnecting wires** (routing).
4. **Respect the given interconnection pattern** (also called the *netlist*).

Placement assumes that the layout of each individual subentity (like a NAND gate) is already known.



📦 Realistic Placement Models

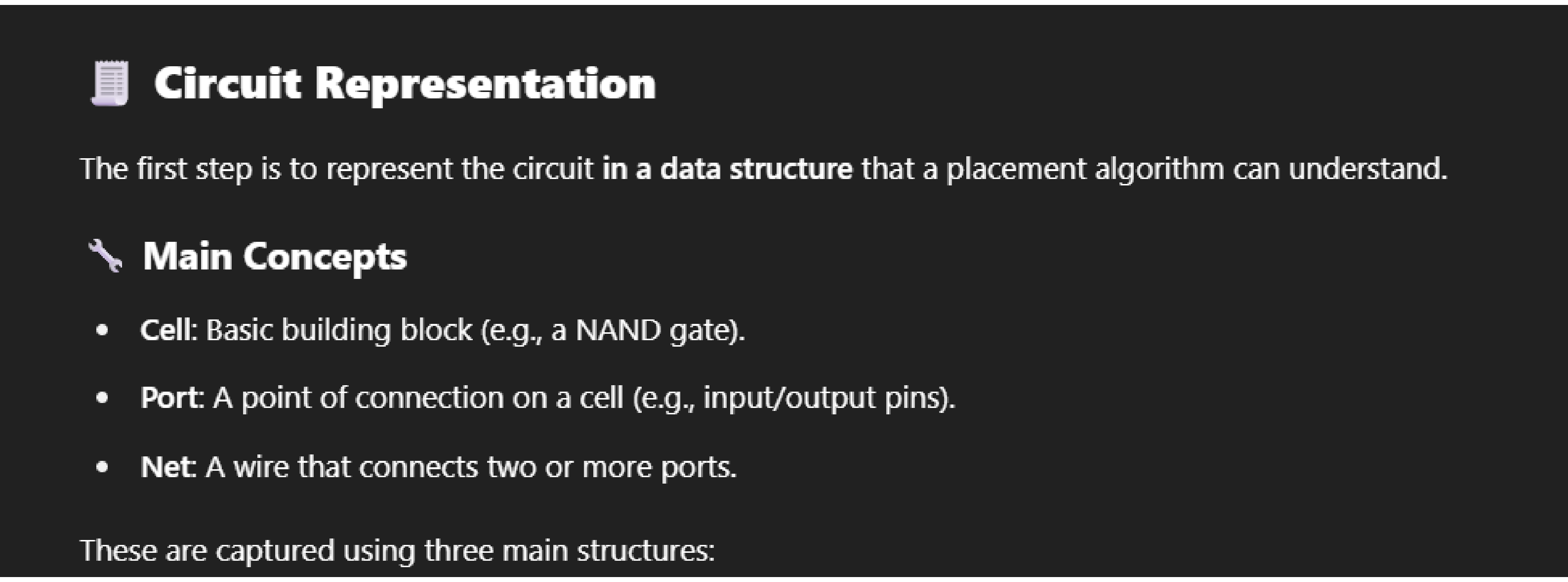
Earlier, a simplified "unit-size placement" problem was introduced. Now, more realistic models are considered:

- **Standard-cell placement**: All cells have the same height but variable width, arranged in rows.
- **Building-block placement**: Cells vary in both height and width (used in more complex, custom layouts).

Most optimization techniques (like Simulated Annealing or Genetic Algorithms) can be applied to both.

🧠 Types of Placement Algorithms

1. **Constructive algorithms**: Build a solution from scratch, placing one cell at a time.
2. **Iterative algorithms**: Start with an initial solution and gradually improve it.



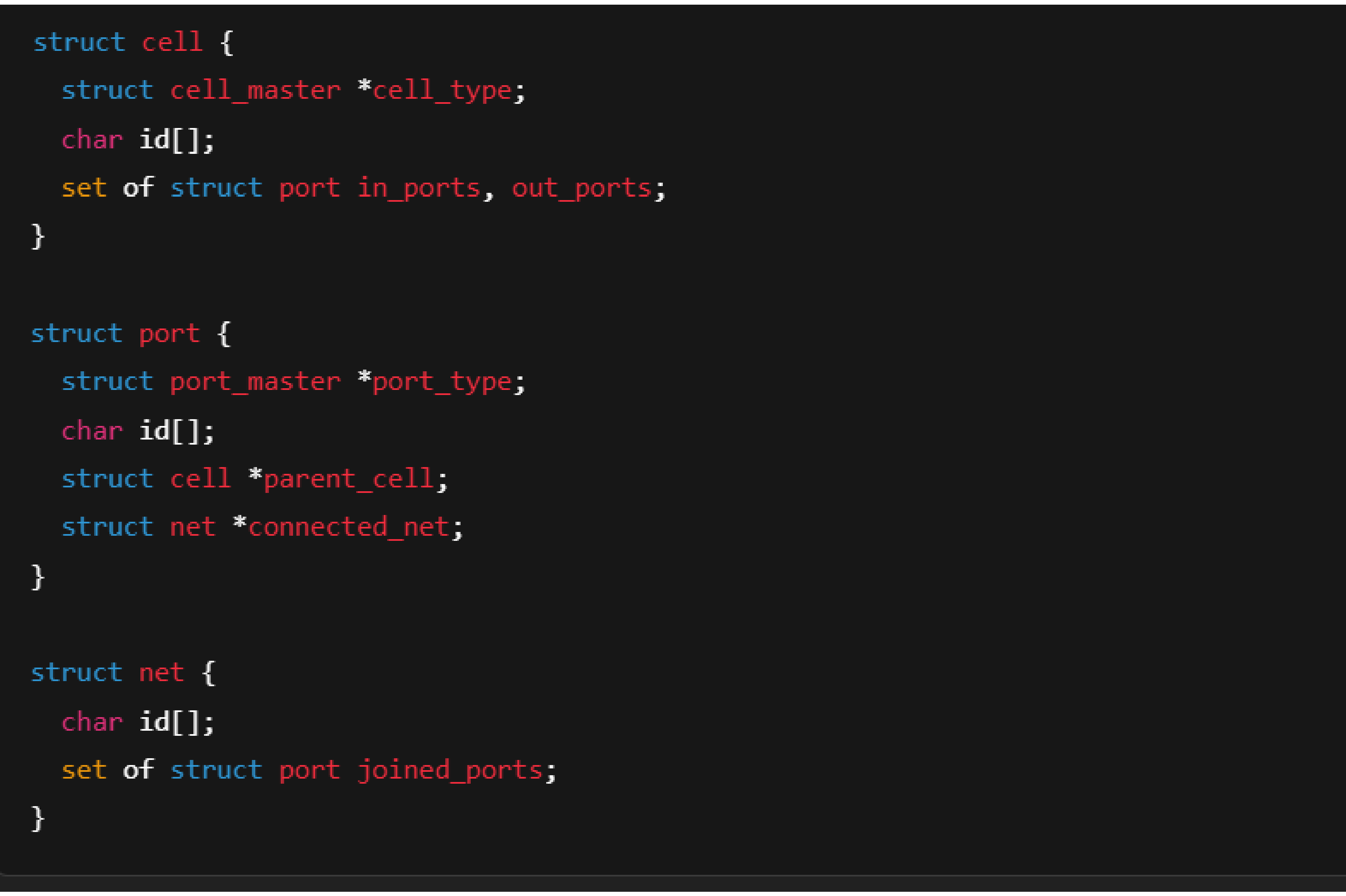
🧾 Circuit Representation

The first step is to represent the circuit **in a data structure** that a placement algorithm can understand.

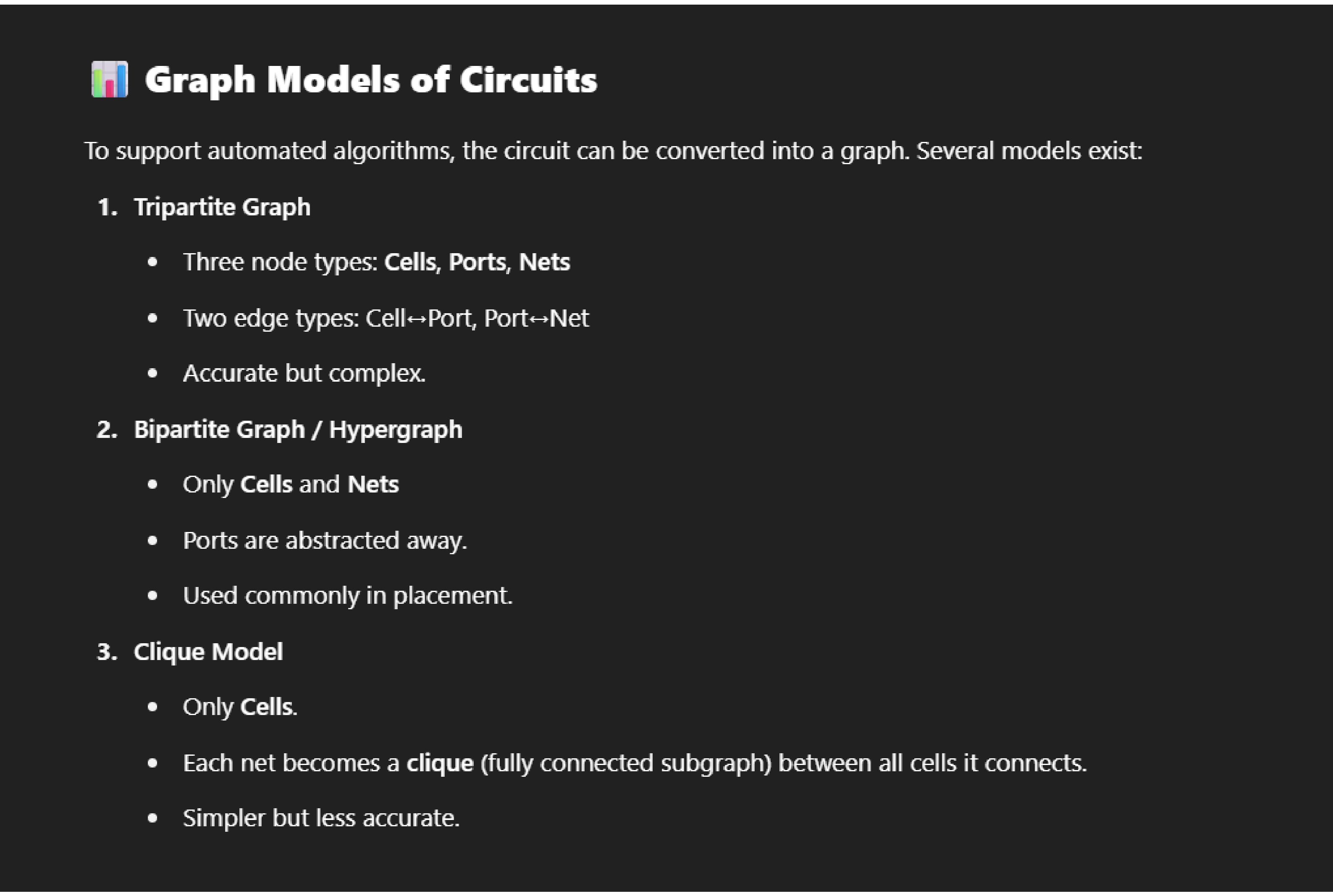
🔧 Main Concepts

- **Cell**: Basic building block (e.g., a NAND gate).
- **Port**: A point of connection on a cell (e.g., input/output pins).
- **Net**: A wire that connects two or more ports.

These are captured using three main structures:



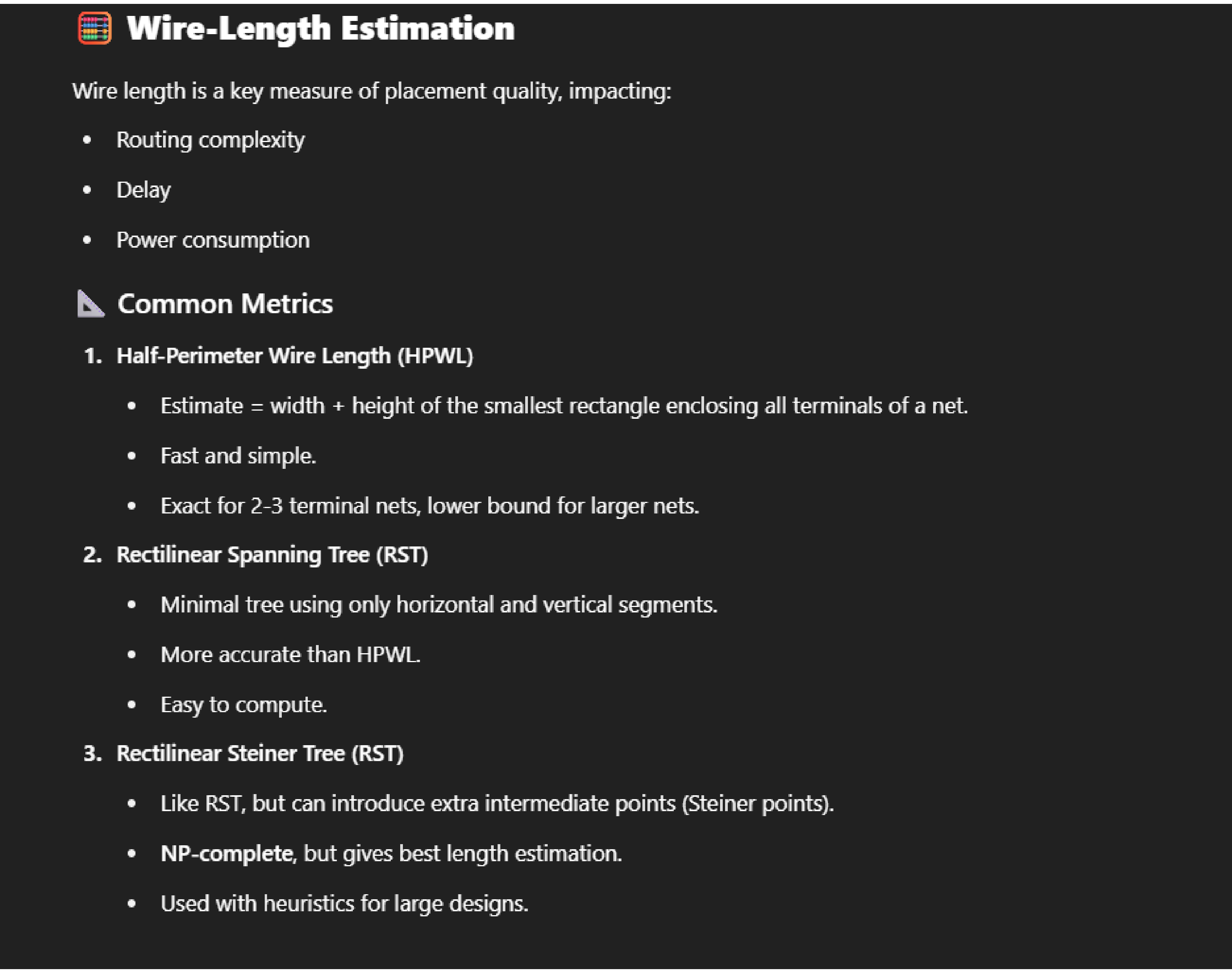
```
struct cell {
  struct cell_master *cell_type;
  char id[];
  set of struct port in_ports, out_ports;
}

struct port {
  struct port_master *port_type;
  char id[];
  struct cell *parent_cell;
  struct net *connected_net;
}

struct net {
  char id[];
  set of struct port joined_ports;
}
```



📊 Graph Models of Circuits

To support automated algorithms, the circuit can be converted into a graph. Several models exist:

1. **Tripartite Graph**
    - Three node types: **Cells, Ports, Nets**
    - Two edge types: Cell↔Port, Port↔Net
    - Accurate but complex.
2. **Bipartite Graph / Hypergraph**
    - Only **Cells** and **Nets**
    - Ports are abstracted away.
    - Used commonly in placement.
3. **Clique Model**
    - Only **Cells**.
    - Each net becomes a **clique** (fully connected subgraph) between all cells it connects.
    - Simpler but less accurate.



🧮 Wire-Length Estimation

Wire length is a key measure of placement quality, impacting:

- Routing complexity
- Delay
- Power consumption

📐 Common Metrics

1. **Half-Perimeter Wire Length (HPWL)**
   - Estimate = width + height of the smallest rectangle enclosing all terminals of a net.
   - Fast and simple.
   - Exact for 2-3 terminal nets, lower bound for larger nets.
2. **Rectilinear Spanning Tree (RST)**
   - Minimal tree using only horizontal and vertical segments.
   - More accurate than HPWL.
   - Easy to compute.
3. **Rectilinear Steiner Tree (RST)**
   - Like RST, but can introduce extra intermediate points (Steiner points).
   - **NP-complete**, but gives best length estimation.
   - Used with heuristics for large designs.

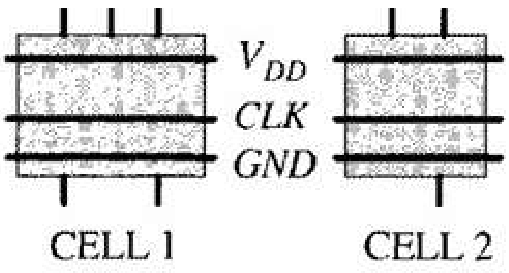


Figure 7.4 Standard cells: logistic signals like power and clock cross the cell at fixed positions while I/O signals specific to a cell have terminals at the top and bottom.

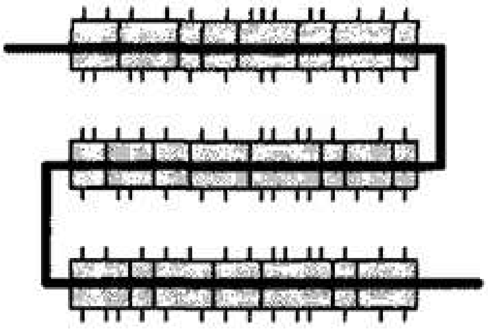
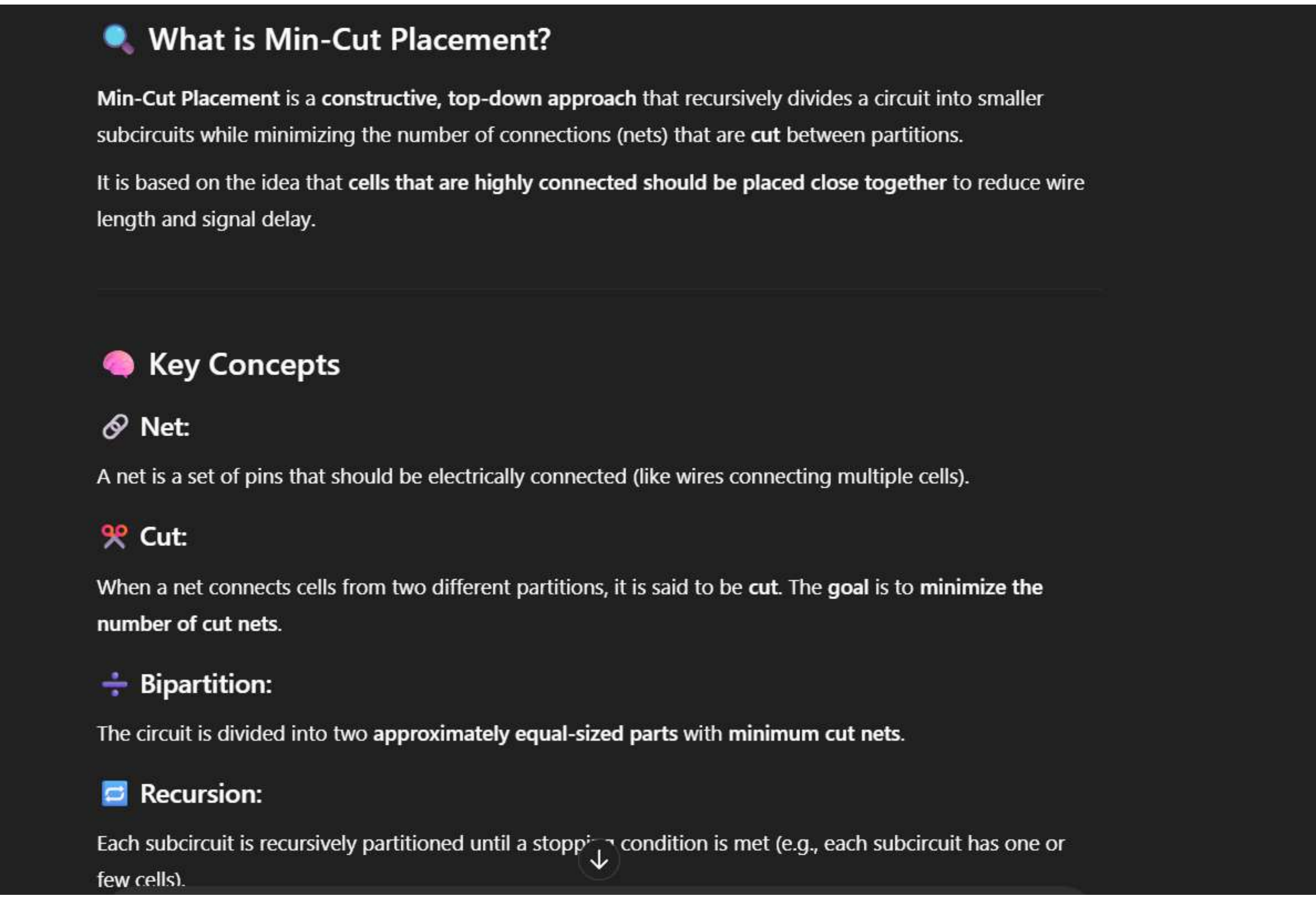


Figure 7.5 An example of standard cell placement: there are three rows separated by wiring channels, while the logistic signals connecting to all cells have symbolically been represented by a thick line.



🔍 What is Min-Cut Placement?

Min-Cut Placement is a **constructive, top-down approach** that recursively divides a circuit into smaller subcircuits while minimizing the number of connections (nets) that are **cut** between partitions.

It is based on the idea that **cells that are highly connected should be placed close together** to reduce wire length and signal delay.

🧠 Key Concepts

🔗 Net:

A net is a set of pins that should be electrically connected (like wires connecting multiple cells).

✂️ Cut:

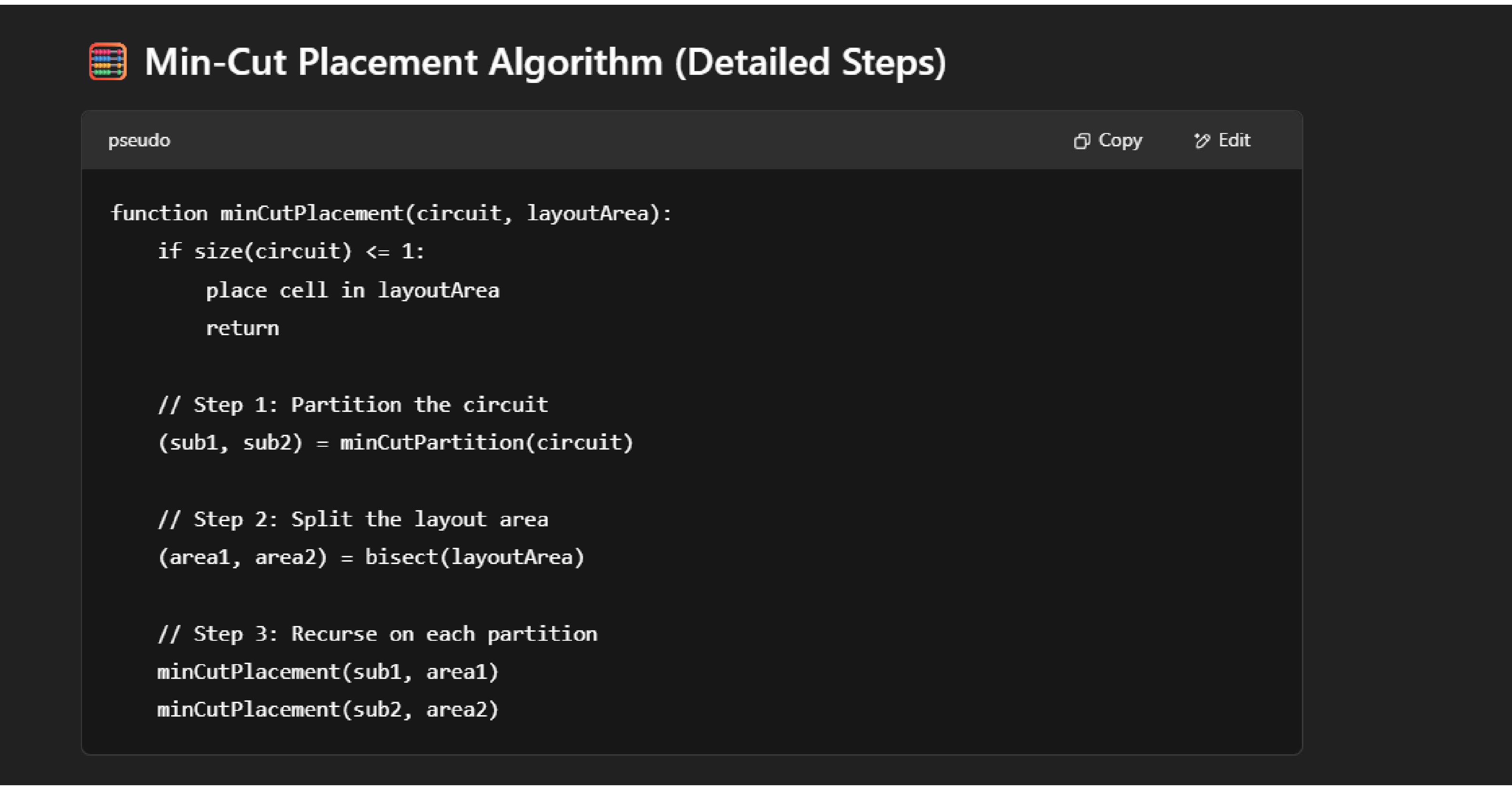
When a net connects cells from two different partitions, it is said to be **cut**. The **goal** is to **minimize the number of cut nets**.

➗ Bipartition:

The circuit is divided into two **approximately equal-sized parts** with **minimum cut nets**.

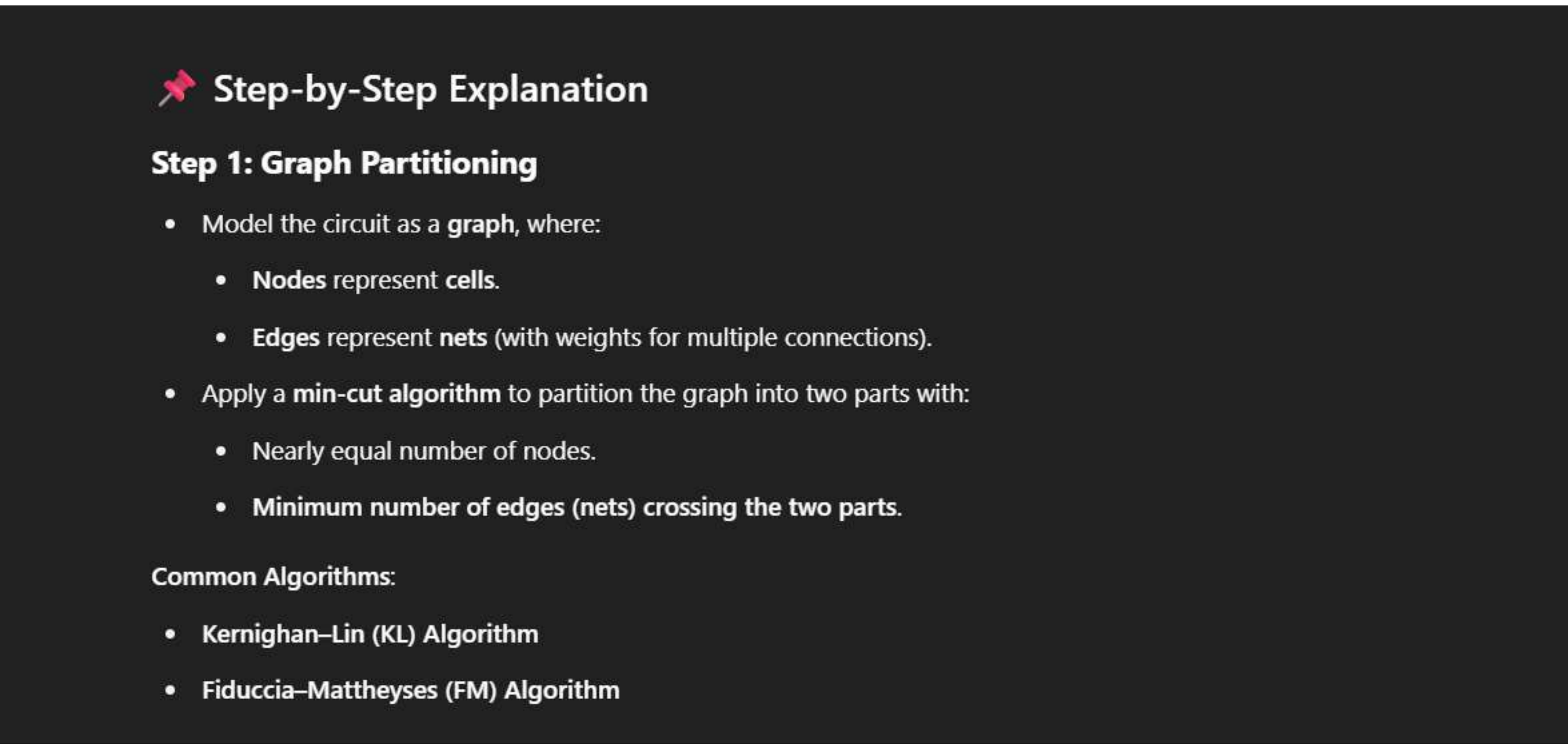
🔁 Recursion:

Each subcircuit is recursively partitioned until a stopping condition is met (e.g., each subcircuit has one or few cells).



Min-Cut Placement Algorithm (Detailed Steps)

```
function minCutPlacement(circuit, layoutArea):
    if size(circuit) <= 1:
        place cell in layoutArea
        return

    // Step 1: Partition the circuit
    (sub1, sub2) = minCutPartition(circuit)

    // Step 2: Split the layout area
    (area1, area2) = bisect(layoutArea)

    // Step 3: Recurse on each partition
    minCutPlacement(sub1, area1)
    minCutPlacement(sub2, area2)
```



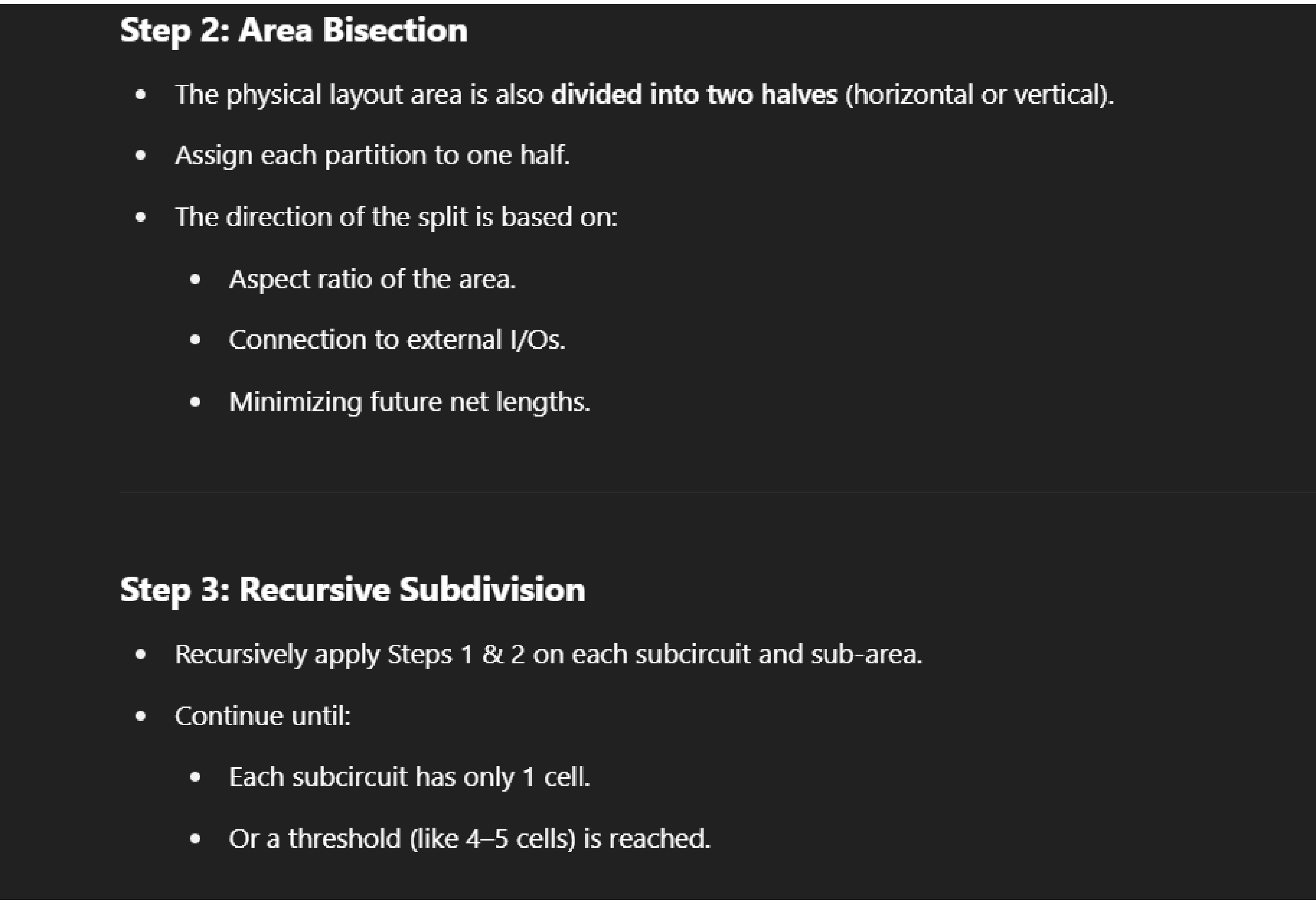
📌 Step-by-Step Explanation

Step 1: Graph Partitioning

- Model the circuit as a **graph**, where:
    - **Nodes** represent **cells**.
    - **Edges** represent **nets** (with weights for multiple connections).
- Apply a **min-cut algorithm** to partition the graph into two parts with:
    - Nearly equal number of nodes.
    - **Minimum number of edges (nets) crossing the two parts**.

Common Algorithms:

- **Kernighan–Lin (KL) Algorithm**
- **Fiduccia–Mattheyses (FM) Algorithm**

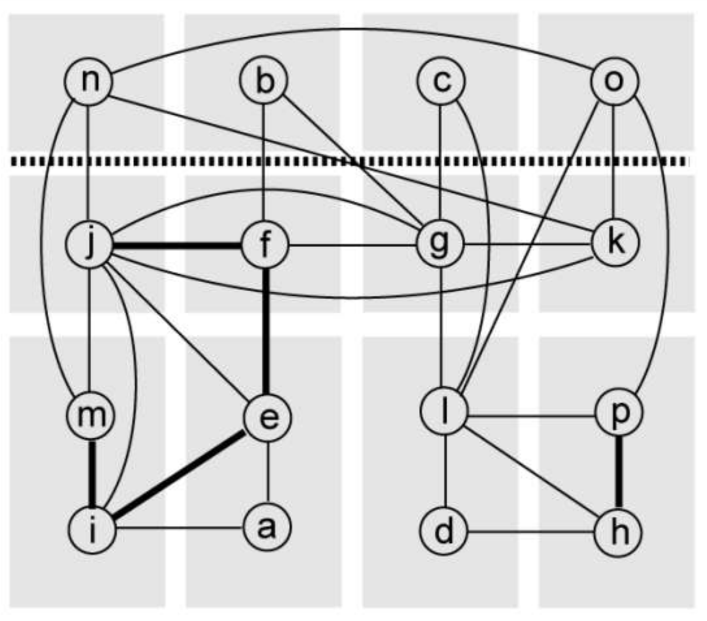


Step 2: Area Bisection

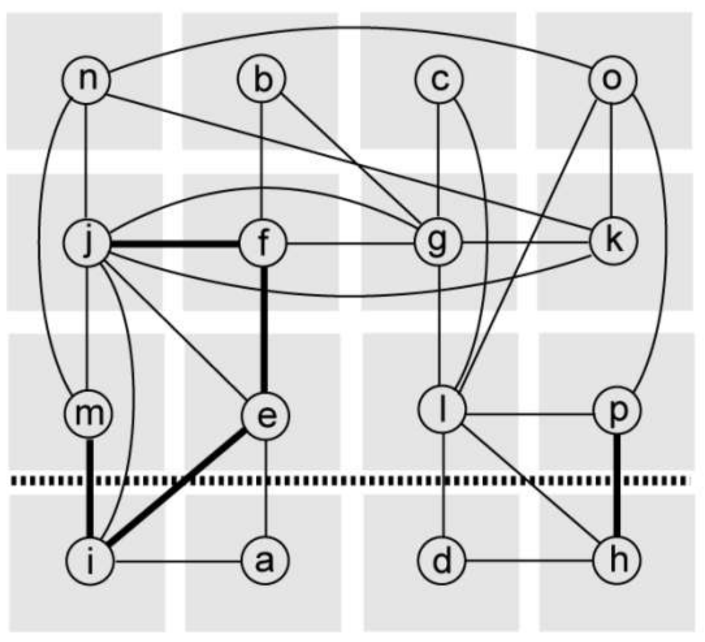
- The physical layout area is also **divided into two halves** (horizontal or vertical).
- Assign each partition to one half.
- The direction of the split is based on:
  - Aspect ratio of the area.
  - Connection to external I/Os.
  - Minimizing future net lengths.

Step 3: Recursive Subdivision

- Recursively apply Steps 1 & 2 on each subcircuit and sub-area.
- Continue until:
  - Each subcircuit has only 1 cell.
  - Or a threshold (like 4–5 cells) is reached.

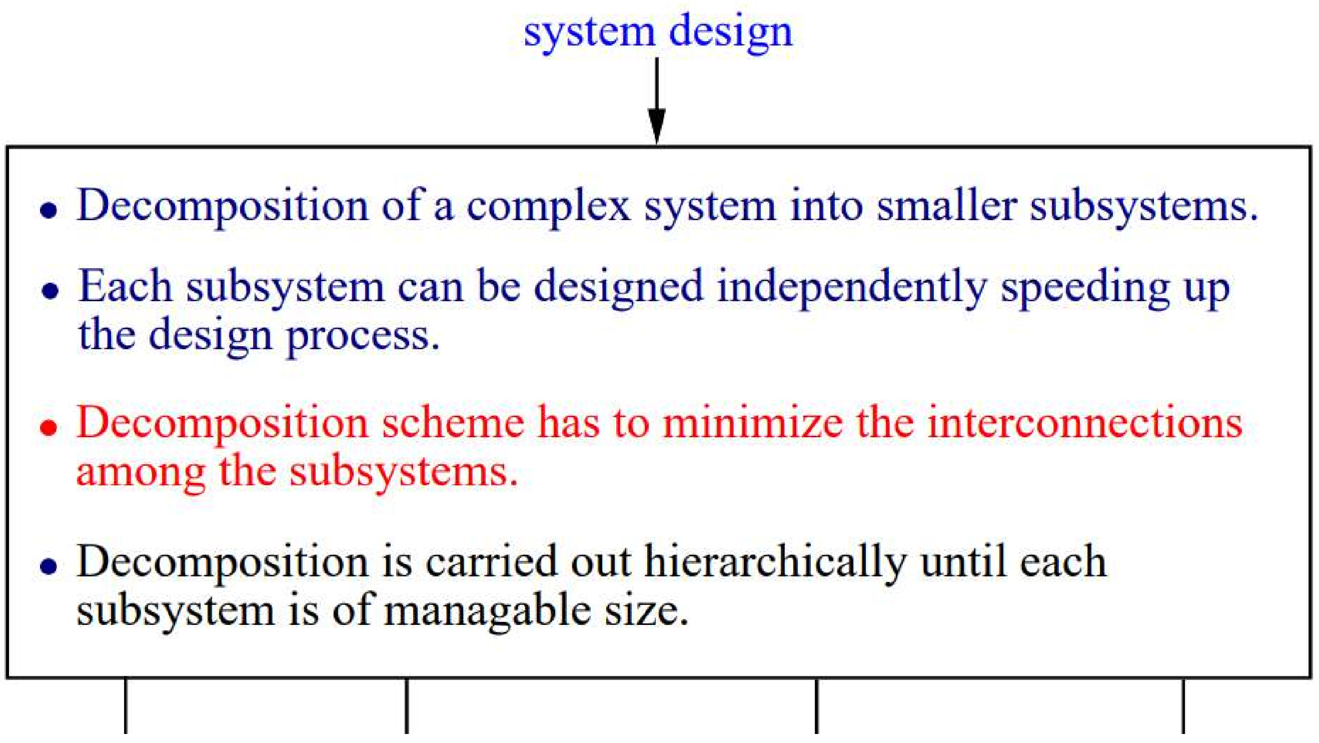


(e) cut 5



(f) cut 6, 2nd-level quadrants formed

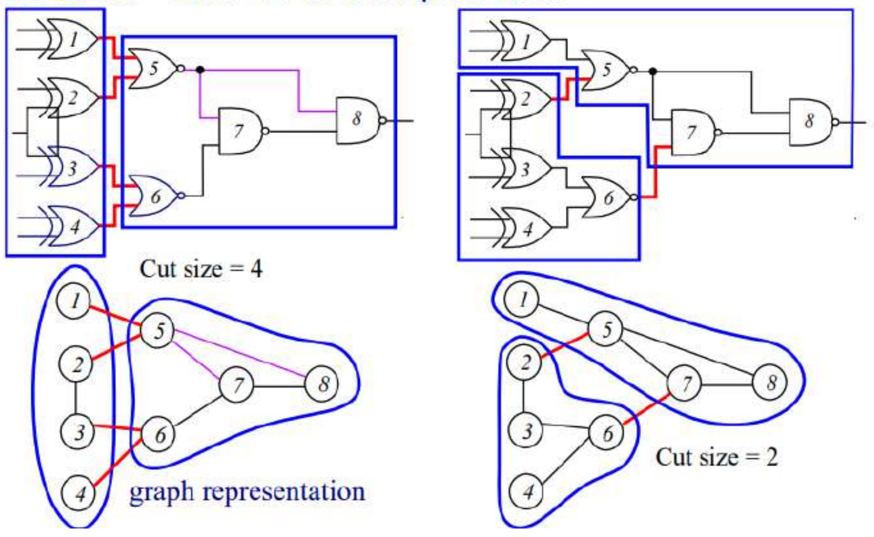
Partitioning



- Decomposition of a complex system into smaller subsystems.
- Each subsystem can be designed independently speeding up the design process.
- Decomposition scheme has to minimize the interconnections among the subsystems.
- Decomposition is carried out hierarchically until each subsystem is of managable size.

Circuit Partitioning

- **Objective:** Partition a circuit into parts such that every component is within a prescribed range and the # of connections among the components is minimized.
  - More constraints are possible for some applications.
- Cutset? Cut size? Size of a component?

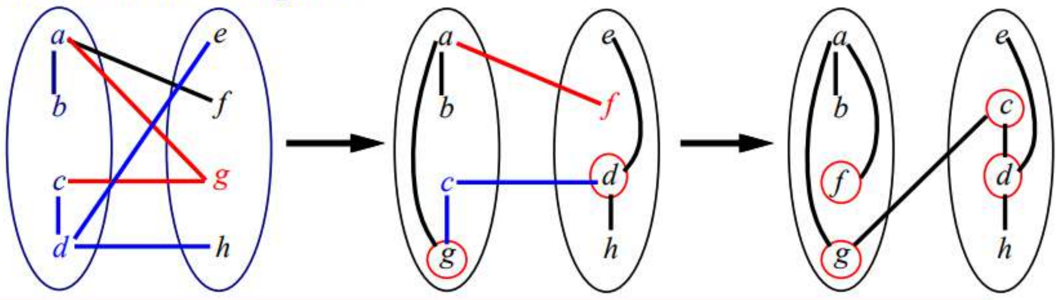


Kernighan-Lin Algorithm

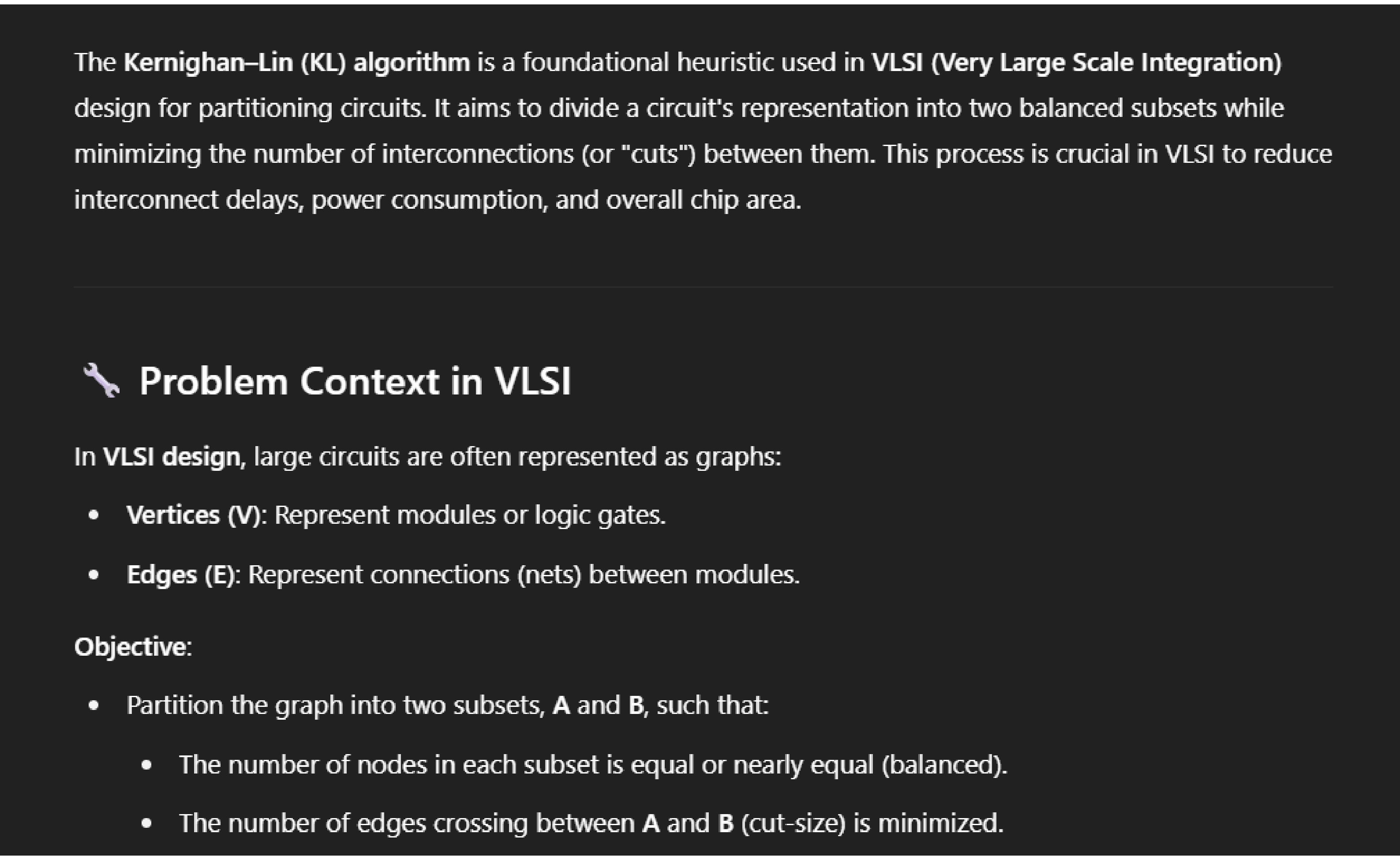
- Kernighan and Lin, "An efficient heuristic procedure for partitioning graphs," The Bell System Technical Journal, vol. 49, no. 2, Feb. 1970.
- An **iterative**, **2-way**, **balanced** partitioning (bi-sectioning) heuristic.
- Till the cut size keeps decreasing
  - Vertex pairs which give the largest decrease or the smallest increase in cut size are exchanged.
  - These vertices are then **locked** (and thus are prohibited from participating in any further exchanges).
  - This process continues until all the vertices are locked.

Kernighan-Lin Algorithm: A Simple Example

- Each edge has a unit weight.



Step #	Vertex pair	Cost reduction	Cut cost
0	-	0	5
1	{d, g}	3	2
2	{c, f}	1	1
3	{b, h}	-2	3
4	{a, e}	-2	5



The **Kernighan–Lin (KL) algorithm** is a foundational heuristic used in **VLSI (Very Large Scale Integration)** design for partitioning circuits. It aims to divide a circuit's representation into two balanced subsets while minimizing the number of interconnections (or "cuts") between them. This process is crucial in VLSI to reduce interconnect delays, power consumption, and overall chip area.

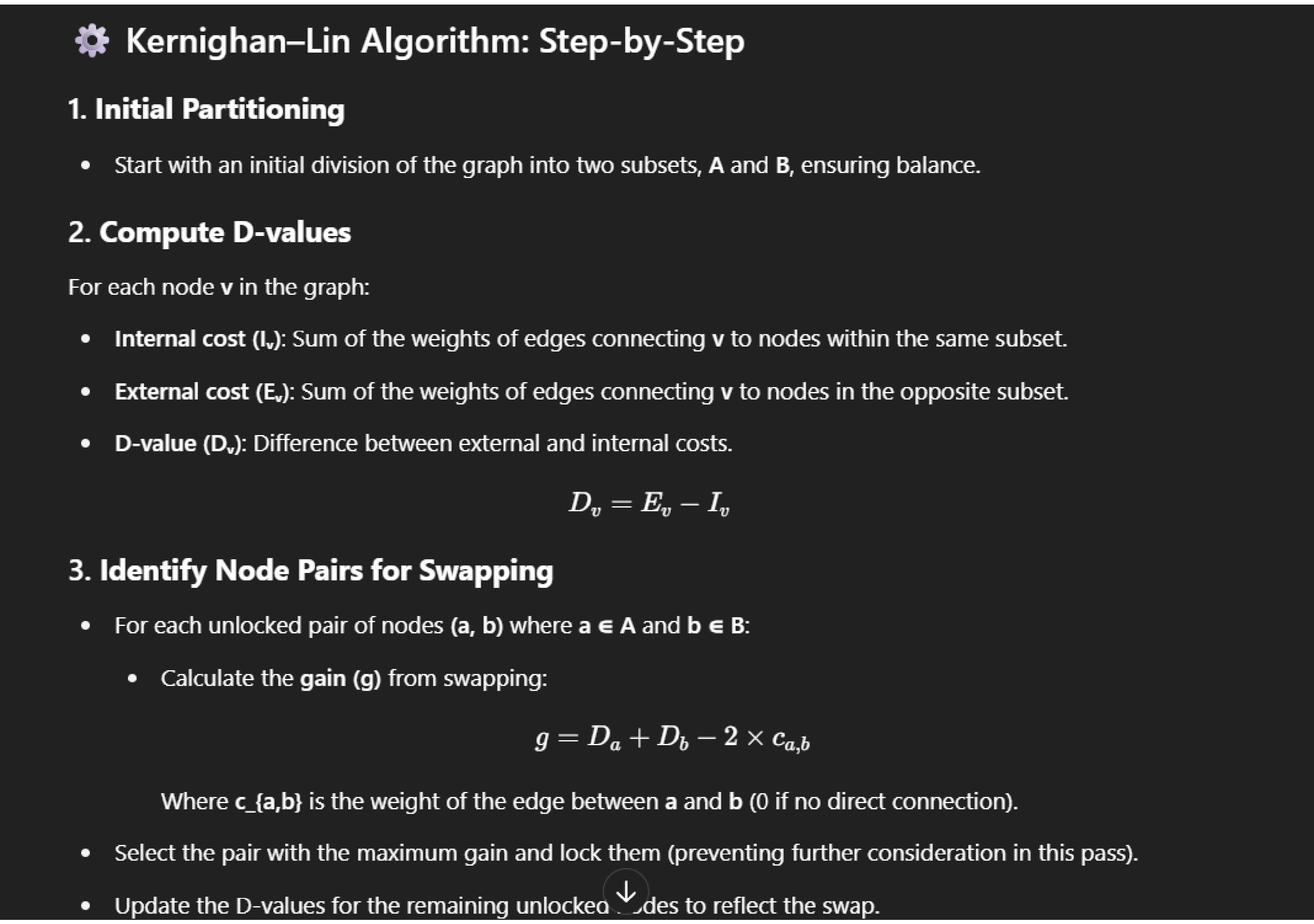
🔧 Problem Context in VLSI

In **VLSI design**, large circuits are often represented as graphs:

- **Vertices (V)**: Represent modules or logic gates.
- **Edges (E)**: Represent connections (nets) between modules.

Objective:

- Partition the graph into two subsets, **A** and **B**, such that:
    - The number of nodes in each subset is equal or nearly equal (balanced).
    - The number of edges crossing between **A** and **B** (cut-size) is minimized.



Kernighan–Lin Algorithm: Step-by-Step

1. Initial Partitioning

- Start with an initial division of the graph into two subsets, **A** and **B**, ensuring balance.

2. Compute D-values

For each node **v** in the graph:

- **Internal cost (I_v)**: Sum of the weights of edges connecting **v** to nodes within the same subset.
- **External cost (E_v)**: Sum of the weights of edges connecting **v** to nodes in the opposite subset.
- **D-value (D_v)**: Difference between external and internal costs.

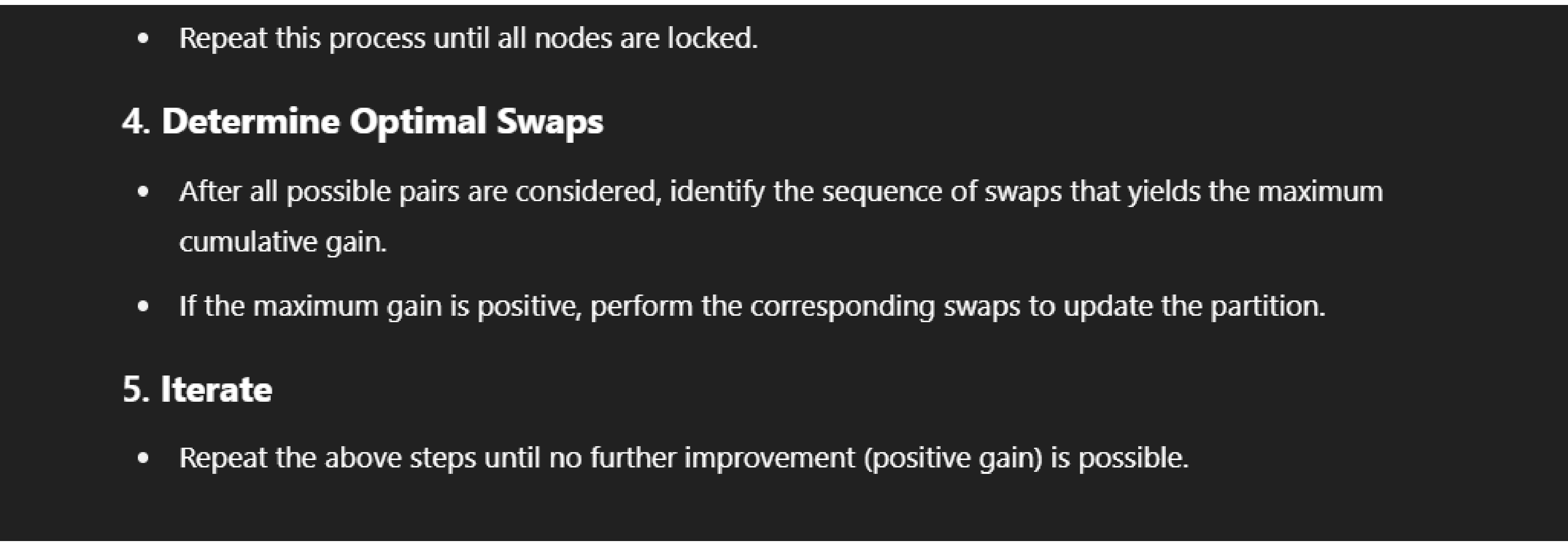
$$D_v = E_v - I_v$$

3. Identify Node Pairs for Swapping

- For each unlocked pair of nodes **(a, b)** where **a ∈ A** and **b ∈ B**:
  - Calculate the **gain (g)** from swapping:

    $$g = D_a + D_b - 2 \times c_{a,b}$$

    Where **c_{a,b}** is the weight of the edge between **a** and **b** (0 if no direct connection).
- Select the pair with the maximum gain and lock them (preventing further consideration in this pass).
- Update the D-values for the remaining unlocked [illegible]des to reflect the swap.



- Repeat this process until all nodes are locked.

4. Determine Optimal Swaps

- After all possible pairs are considered, identify the sequence of swaps that yields the maximum cumulative gain.
- If the maximum gain is positive, perform the corresponding swaps to update the partition.

5. Iterate

- Repeat the above steps until no further improvement (positive gain) is possible.

Time Complexity

- Each pass involves evaluating all possible pairs, leading to a time complexity of **$O(n^2)$** per pass, where **n** is the number of nodes.
- The number of passes depends on the specific graph and desired optimization level.