ChatGPT

自動レイアウト／配線アルゴリズム

デジタル回路図の自動配置・配線は、グラフ描画の典型的な問題とみなせます。代表的な手法として、**力学モデルに基づくレイアウト**（スプリング埋め込みなど）や**階層型（Sugiyama）レイアウト**、**直交レイアウト**、**最適化手法（焼きなまし法など）**があります。これらを組み合わせ、入出力やゲートの接続を美しく示します。たとえば、力学モデルでは「ノード間にバネ（引力）と斥力を置き、シミュレーションでエネルギーを最小化する」ことで対称性のある配置を得ます [1]。この手法は小規模（数十〜百程度）のグラフで美的で交差の少ない配置を生成できますが、大規模では計算量や局所解の問題で効果が低下します [2]。一方、**Sugiyama方式**の階層型レイアウトは、有向グラフ（デジタル回路の信号フローなど）を水平レイヤーに分割して描画します。Graphvizの `dot` などがこの手法を用いており、ノードを層に割り振り、層間の結線交差を減らすように順序入替やダミーノードを導入します [3]。有向非巡回グラフに適し、信号の流れを視覚的に表現しやすい反面、レイヤー数やノード幅制約の最適化はNP困難であり、ヒューリスティックに頼る場合が多いです [4] [3]。**直交レイアウト**ではエッジを水平／垂直線分で表現し、折れ点数やエリアを最小化します。Di Battista・Tamassiaらの Giotto アルゴリズムは「平面グラフ上でエッジ数を最小曲げで描画する」手法として知られ、実装例では各辺の屈曲点を流れに沿って最適化します [5]。ただし直交化やコンパクションには計算コストがかかり、中規模以上では高速化やマルチレベル化が必要です。**焼きなまし法**などの組合せ最適化手法も使われ、EDAではセル配置やフロアプランニング、配線長最小化で定評があります [6]。焼きなまし法は確率的に配置を改良し、配線長や交差数のグローバル最適化に寄与しますが、計算負荷とパラメータ調整の難しさがあります。

- **力学モデル（Force-directed）**: ノードにバネや反発力を仮定し反復配置。美的で交差の少ない結果を生成しやすい [1]。少数百ノード程度まで有効だが、大規模化に弱く、多段階化（マルチレベル手法）で拡張する例がある [2]。
- **階層型（Sugiyama）**: 有向無巡回グラフに適用。ノードを水平層に割り付け、層間で交差を減らすよう調整 [3]。業界標準（Graphviz dot など）で用いられる。ノードの有向性を活かせるが、方向性のないグラフでは適用前に仮想的に向きを付ける必要がある（フィードバックアークセット問題）。
- **直交レイアウト**: エッジを水平／垂直線分で描き、屈曲点やエリア最小化を目指す。Giotto（Tamassia）などが有名で、平面グラフの最小屈曲直交描画を実現する。直交レイアウトは回路図でよく用いられるが、非平面グラフには図を追加補正するか、交差許容する必要がある。
- **最適化（焼きなまし・遺伝的）**: 配置や配線長などのコストを評価関数とし、試行錯誤的に最適化。VLSI配置・配線で実績があり、EDAの配置・配線問題に応用されている [6]。高品質な結果を得やすいが、計算時間が膨大になりやすく、ヒューリスティック設計や初期配置が重要。

配線アルゴリズム

回路図内の配線（ネットの接続経路探索）は、典型的にはグリッド上の最短経路問題として扱われます。**迷路法（Lee法）**は幅優先探索（BFS）で波面を広げて最短経路を見つける手法で、解が存在すれば必ず最短路を保証します [7]。具体的には、スタートから全方向に距離ラベルをつけて波形展開し、目標到達後に逆追跡でルートを決定します [7]。この手法はシンプルで確実ですが、グリッドサイズが大きくなるとメモリ・時間消費が増大します。**線分探索法（Line Search）**は、水平線→垂直線と交互に直線探索を行う古典的アルゴリズムで、工学的には「水平に伸びる線分から次に垂直線分へ」と続け、目標に到達したら経路を確定します。これは早期の自動配線で提案された方法で、最短路保証はなく「短く屈曲少なめの経路を選ぶ」実装が一般的です [8]。さらに効率化するため、**A**探索やダイクストラ法でヒューリスティックを利用する例も多く、特に回路パッド間が格子点にな

らない連続空間では有効です。複数ピンにまたがるネットでは、*Steiner*木*アプローチで合流点を作り総配線長を減らすことも考えられます。

- **迷路ルータ（Leeのアルゴリズム）**: マンハッタン格子上でBFSを行い、最短経路を求める。必ず最適解を返すが、多数ネットを個別に処理すると全体最適にはならないことや、計算・メモリが増加する課題がある [7]。
- **線分探索（Line Routing）**: スタート地点から水平方向に直線を伸ばし、次に垂直線を伸ばすのを繰り返す手法 [8]。探索終了後に最短・最少屈曲路を選ぶ工夫が必要。過去の自動配線で古典的に使われた方式で、実装は比較的単純だが複数経路の比較や大規模ネットには不向き。
- **X-Y（チャネル）ルーティング**: PCBなどで使われる手法で、複数層を用い、水平配線と垂直配線を別層に分離して進める。これにより「間違った方向（wrong-way）配線によるボトルネックを防ぎ、後続配線の自由度を保つ」メリットがある [9]。逆に、層数が必要である点や単層板には使えない点がある。
- **A*/最短路探索**: セル単位のグリッドをグラフとみなし、ゴールまでのマンハッタン距離などをヒューリスティックに使って探索幅を削減する。迷路法より早い場合が多い。
- **Steiner木/木配線**: ネットに複数ピンがある場合、合流点（ステイナー点）を追加して配線長を短縮。NP困難な問題だが、近似手法やヒューリスティックで扱う。

EDAツール・ライブラリの手法

既存のEDA・レイアウトライブラリでは、上記の手法が実装例やオプションとして見られます。たとえば **Graphviz** は無向/有向グラフの自動描画ツールで、`dot`（Sugiyama 階層法）や `neato`（力学モデル）、`twopi`（放射状）など複数のレイアウトエンジンを持ちます。実際に Graphviz の `dot` は Sugiyama フレームワークを用いており、多くの他のライブラリ（da Vinci、AGD など）もこれに倣っています [3]。Python では NetworkX や Graph-tool などがスプリング埋め込みやスペクトル法を実装しており、NetworkX の `spring_layout` 関数で力学モデル配置、`graphviz_layout` 関数で Graphviz 呼び出しが可能です。また直交レイアウト用には OGDF（Open Graph Drawing Framework、C++）などに Giotto アルゴリズムの実装があります。

PCB設計向けには、**KiCad** では外部ルータ（FreeRouting 等）経由のオートルータ機能が利用可能で、内部でもレイヤーごとの優先方向設定やプッシュ＆シューブ（配線ぶつかり時に押しのけ）による配線が提供されています。**Eagle** には TopoRouter（Topology ルータ）というプリルーター機能もあり、これは優先方向を持たない「トポロジカル」ルータとして知られます。これら市販ツールの具体的アルゴリズム詳細は公開されていませんが、基本的には迷路＋最適化や層別配線の組合せです。

YAMLデータからの変換とワークフロー

YAML に記述された回路構造（論理ゲート・入出力・ネットリストなど）を図面化する流れの例は以下の通りです。まず YAML をパースして回路を**グラフモデル**（ノード＝部品/ピン、エッジ＝ネット）に変換します。Python では `PyYAML` で読み込み、NetworkX のグラフとして構築する例があります。次にグラフレイアウトを選択し実行します。小規模なら `nx.spring_layout`（力学モデル）、有向で階層図なら `nx.nx_agraph.graphviz_layout`（dot 経由）などを使います。得られたノードの位置（座標）に基づき、次に配線経路を計算します。配線層を仮想格子にモデル化し、各ネットについて例えばLee アルゴリズムや A によって経路を求めます。障害物（既配線や部品領域）を考慮しつつグリッド上で波面展開を行い、到達点から逆向きに経路復元します。このように自前で経路を得た後、各線分を曲線や折れ線で表現します。最後に出力形式へ変換します。*EDA*出力では、*KiCad* なら `.net` や `.kicad_sch` 形式のデータを生成し、コンポーネントの座

標・ネットリストを埋め込むことで*Eschema*に取り込めます。画像出力*では Graphviz であれば直接 SVG/PNG を作成できますし、Matplotlib や Cairo でパスを描くことも可能です。

- **ステップ例**（YAML→図示）: YAMLパース → グラフ構築 → レイアウト（位置計算） → 各ネット配線経路計算 → 出力ファイル生成。
- **ライブラリ例**: Python では `networkx`（グラフ・レイアウト）、`pyyaml`（YAML）、`matplotlib`/`svgwrite`（描画）、あるいは `subprocess` 経由で `dot`（Graphviz）呼び出しなどを組み合わせる。

実装例（擬似コード）

以下に Python での実装スケッチ例を示します。YAMLから部品とネット情報を取得し、NetworkX グラフに登録、Springレイアウトで配置し、迷路法で配線する例です（実運用にはエラー処理や細部調整が必要です）。

```
import yaml
import networkx as nx
from collections import deque

# YAML読み込み例（部品とネットリストの想定フォーマット）
with open('circuit.yaml') as f:
    data = yaml.safe_load(f)
G = nx.Graph()
# ノード（部品ID）追加
for comp in data['components']:
    G.add_node(comp['id'], type=comp['type'])
# エッジ（ネット）追加
for net in data['nets']:
    pins = net['pins']  # 例: ['U1.1', 'U2.3', ...]
    # 単純化のため、完全グラフで繋ぐ例（実際はスター接続等で枝分かれさせる）
    for i in range(len(pins)):
        for j in range(i+1, len(pins)):
            G.add_edge(pins[i], pins[j])

# レイアウト計算（力学モデル配置）
pos = nx.spring_layout(G, k=0.5)  # 得られたposはノード毎の座標

# 単純な迷路ルート関数（グリッド上のBFS）
def maze_route(start, end, obstacles):
    queue = deque([start])
    parent = {start: None}
    while queue:
        cur = queue.popleft()
        if cur == end:
            break
        x,y = cur
        for dx,dy in [(1,0),(-1,0),(0,1),(0,-1)]:
```

```
            nxt = (x+dx, y+dy)
            if nxt in obstacles or nxt in parent: continue
            parent[nxt] = cur
            queue.append(nxt)
    # 経路復元
    path = []
    node = end
    while node:
        path.append(node)
        node = parent.get(node)
    return list(reversed(path)) if path[-1] == start else None

# 障害物（部品領域や既配線）を定義（例として空リスト）
obstacles = set()
# 各ネットの配線例（ここでは2ピンのみのネット想定）
routes = {}
for net in data['nets']:
    if len(net['pins']) == 2:
        p1, p2 = net['pins']
        # 座標系に合わせてグリッド点に変換（例: roundを使う）
        s = (round(pos[p1][0]), round(pos[p1][1]))
        t = (round(pos[p2][0]), round(pos[p2][1]))
        path = maze_route(s, t, obstacles)
        routes[(p1,p2)] = path
        # 経路点を障害物に追加しない場合は重なりありうる
```

上記ではごく基本的な構造を示しました。実際には、ピン情報からワイヤ用グリッドを生成し、層や方向制約も考慮して効率よくルーティングする必要があります。また、部品配置では複数のアルゴリズム（スプリング埋め込みと階層型の併用、または手動調整）や、配線ではステイナー木やRip-up&Rerouteのような改善手法を加えることで品質を高められます。

以上のように、**力学モデル、階層配置、直交ルート、迷路探索**などの既存アルゴリズムを理解し、YAML→グラフ→レイアウト→配線というパイプラインに組み込むことで、自動回路図生成システムを実装できます。各手法の特性（スケーラビリティ、最適性、制約条件）は前述の通りであり、規模や用途に合わせて適切な組合せを選ぶことが重要です [1] [7]。

参考資料: 上記手法や実装例については、グラフ描画の文献やEDA関連資料で詳述されています [1] [3] [7] [6]。各アルゴリズムの具体的なコード例やライブラリ呼び出しは、使用環境に応じて選択してください。

---

[1] [2] main.dvi
https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/force-directed.pdf

[3] cs.brown.edu
https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/hierarchical.pdf

4 Layered graph drawing - Wikipedia
https://en.wikipedia.org/wiki/Layered_graph_drawing

5 alg_patterns.dvi
https://cs.brown.edu/cgc/jdsl/papers/alg_patterns.pdf

6 Simulated Annealing : Methods and Real-World Applications – OMSCS 7641: Machine Learning
https://sites.gatech.edu/omscs7641/2024/02/19/simulated-annealing-methods-and-real-world-applications/

7 Lee algorithm - Wikipedia
https://en.wikipedia.org/wiki/Lee_algorithm

8 第64回 プログラミングについて『迷路を通ろう！（その１）』 - アポロレポート
https://www.apollo-g.co.jp/blog/column/a130

9 PCB Routing Methods | TERRATEL
https://www.terratel.eu/routing-methods.html