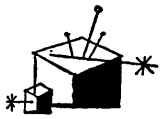
講　座



最近のグラフ理論とその応用 (4)†

大　附　辰　夫††

6. グラフ理論の応用

グラフ理論の応用を論ずる際には，先ず Kirchhoff (1824—87) の昔にさかのぼる電気回路とグラフとの切り離すことのできない縁を認識する必要がある．実際，システムのグラフ的構造の記述という抽象的手段に留まらず，方程式の定式化および解析のための具体的手段を提供しているという意味において，グラフ理論の一大応用分野として電気回路理論を挙げなければならない．電気回路の解析において，本講座の3で論じたグラフの代数的構造が本質的な役割を演じており，この立場からの記述は多くの文献の中に見い出される[1),11),59)]．

1960年代に入ると，グラフ理論が輸送計画，日程計画，通信網などの工学的応用と関連して広く研究されるようになり，現在ではネットワーク型の計画問題に対する系統的な理論が確立されている[4),5),8),10)]．この分野においては，本講座の4で論じたように，グラフ理論が統一的議論を可能にし，有効な解析手段を提供している．

1970年代から，計算機の進歩や集積回路技術の発展に刺激を受けて，大規模システムを解析するためのスパース行列処理技法[60),61)]，LSI やプリント基板の設計自動化[62)]などの新しい研究課題が登場したが，これらの分野もグラフ理論の主要な応用分野とみなされる．特に配置配線設計問題においては，部分問題として多くのグラフの問題が抽出され，最近のグラフ理論の研究の動機となっている．一方では計算の複雑度の理論（本講座の5参照）の発展により，配置配線設計に関連した問題の殆んどが“難しい”問題に属することも明らかになったが，これも“新しい研究の方向付けを与えた”という意味においてグラフ理論研究の成果の一つと考えられる．

以上代表的なグラフ理論の応用分野を挙げたが，これらについては多くの文献に解説されており，筆者自身の解説も本学会誌に取り上げられている[61),61)]．そこで重複を避ける意味と，議論の焦点を絞る意味を含めて，グラフ理論の応用の対象をコンピュータ・サイエンスの分野に限定して解説を試みる．

実際に技術者が遭遇する解析・設計問題において，グラフ理論が適用できる部分は，主として定性的側面に限られており，詳細な定量的議論をするためには，他の手段を用いなければならない．コンピュータ・サイエンスの分野にグラフ的モデルが導入されるようになった理由は，第1に，グラフが種々多様な問題の構造を統一的に記述する上で便利であるということである．第2にグラフを扱うアルゴリズムがプログラミングシステムの正しさの判定や性能向上の指針を与えるための手段を提供するということである．

6.1 タスク・スケジューリング問題

スケジューリング問題のモデル化手法はオペレーションズリサーチの分野におけるジョブ・ショップの解析に端を発し[63)]，並列処理機能を含むコンピュータシステムの出現によって発展してきたものである．本講座では特にグラフ理論的考察が主体となる**決定性スケジューリング** (deterministic scheduling) に焦点を絞り，いくつかのトピックについて解説する．詳しい議論については文献[64),65)]を参照されたい．

6.1.1 決定性スケジューリング

コンピュータに入力されるジョブをタスクの集合 $T=(T_1, T_2, \cdots, T_n)$ とみなし，各々のタスク T_i の処理時間を τ_i とする．またタスク相互の先行後続の半順序関係は無サイクル*有向グラフ $G=(T, E)$ によって表わされる．すなわち“タスク T_i をタスク T_j より先に処理する”という条件は有向枝 $(T_i, T_j) \in E$ によって表現される．以後無サイクル有向グラフ $G=(T, E)$ と $\tau_i ; i=1, 1, \cdots, n$ の組を持って**タスクシステム**と呼ぶ．

タスクシステムにおける**スケジューリング**とは与え

† Recent Development in Graph Theory and Its Applications by Tatsuo OHTSUKI (Dept. of Electronics and Communication Engineering, School of Science and Engineering, Waseda University).

†† 早稲田大学理工学部電子通信学科

* 有向閉路（サイクルとも呼ばれる）を持たないという意味

```
begin
comment intialization;
    W=φ; i=1;
    for all v∈T do
    begin COUNT(v)=INDEGREE(v);
        if COUNT(v)=0 then W=W∪{v}
    end;
comment main loop;
    while W≠φ do
    begin v=first vertex in W; W=W−{v};
        LABEL (v)=i; i=i+1;
        for all w∈T such that (v, w) ∈E do
        begin COUNT(w)=COUNT(w)−1;
            if COUNT (w)=0 then W=W∪{w}
        end
    end
end
```

図-36　トポロジカル・ソート

られた半順序関係に従ってタスクを並行処理可能なプロセッサ $P_1, P_2, \cdots, P_s$; $s \geqq 2$ に割り当てることであり，次の2つのフェースから成っている．

フェースⅠ　与えられたタスクシステムに対してあるラベリング・アルゴリズムを適用してタスク処理の優先順位を表わす**プライオリティ・リスト**を出力する．

フェースⅡ　各時刻 t において，空いているプロセッサに（半順序関係の意味で）処理可能なタスクの中でプライオリティ・リストにおける優先順位の最も高いものを割り当てる．そのようなタスクが存在しない場合は $t \leftarrow t+\Delta t$ として同じことを繰り返す．ここで Δt はクロックの間隔である．以後一般性を失うことなく，$\Delta t=1, \tau_i (i=1 \sim n)$ は整数と仮定する．

グラフ G において，トポロジカルソート（**図-36**参照）を適用して節点（タスク）にラベル付けすればフェースⅠにおけるプライオリティリストの1つが得られる．すなわち，T_i が T_j に先行するとき LABEL (T_i)<LABEL (T_j) という性質を満たすラベル付けが行われる．図において INDEGREE (v) は節点（タスク）$v \in T$ に入ってくる枝の数を表わし，W は各時点でラベル付け可能になった節点の集合を表わしている．もし W の記憶のためにキュー（スタック）を用いれば横型探索（縦型探索）に基づくトポロジカルソートが行われる．またこのアルゴリズムの漸近複雑度は $O(m+n)$; $m=|E|, n=|T|$ である．

対象とする無サイクル有向グラフにおいて，**入口**（入って来る枝を持たない節点）が1個であると仮定してトポロジカルソートのアルゴリズムを部分的に変更すれば最長径路のアルゴリズムが得られる．すなわち，すべての節点 $u \in T$ に対して，

LONGEST_PATH $(u)=0$

```
begin for all u∈T such that (u, w)∈E do
    LONGES_PATH (w)=max{LONGEST_PATH(w), a(u,w)
                +LONGEST_PATH (u)};
    W=W∪{w}
end
```

図-37　最長径路アルゴリズム

という初期操作を行っておくとして，図-36 の main loop の中の $W=W \cup \{w\}$ という代入文を**図-37** のような操作で置き代えれば，入口から他の各々の節点に至る最長径路が求まる．ここで $a(u, w)$ は枝 $(u, w) \in E$ の長さである．重みが枝ではなく節点の上に与えられる場合や1個の出口を基準とした最長径路問題に対するアルゴリズムもほぼ同様である．最長径路アルゴリズムの処理時間も高々枝の数に比例する程度であり，タスクシステムの（準）最適スケジューリングを求めるために使われることがある．

以上のモデル化に基づいてコンピュータシステムの性能を論ずるに当り，最初に2つの警告をしておく必要がある．第1の警告は「決定性スケジューリングにおいては，直観的に想像しにくい変則現象が起り得る」ということである．すべてのタスクの処理を終了するために要する時間 W は半順序関係 $G=(T, E)$，タスクの処理時間 τ_i，プロセッサの数 s およびラベリングアルゴリズム α の4つの要因によって決まるが，この内の3つを固定して，

- 半順序関係をゆるめる（G のいくつかの枝を除く）
- タスク処理時間 τ_i を減少させる
- プロセッサの数 s を増大させる

のいずれかの変更を行った場合に完了時間 w が増大することがある[66]，ということを注意しておく．ここでは最後の場合についての例を挙げておく．**図-38**(a)に示すタスクシステムに対してあるトポロジカルソートを行えば，$(T_1, T_2, \cdots, T_7)$ というプライオリティ・リストが得られる．$s=2$ としてスケジューリングのフェースⅡを実行すれば，**図-38**(b)のガント・チャートが示すように完了時間は $w=6$ となる（これは明らかに最適である）．ところが $s=3$ とすると，**図-38**(c)のような割り付けが行われ $w=7$ となってしまう．

この種の変則現象は詳細に解析され，下記のような厳密な上限が与えられている[66]．半順序関係 $G=(T, E)$，タスク処理時間 τ_i，プロセッサ（同等とする）の個数 s，，ラベリングアルゴリズム α に対する完了時間を w とする．また4つの要因を，それぞれ，$G'=(T, E')$; $E' \subseteq E$, $\tau_i' \leqq \tau_i (i=1 \sim n)$, s', α' によって置き代えた場合の完了時間を w' とすれば，完了時間の

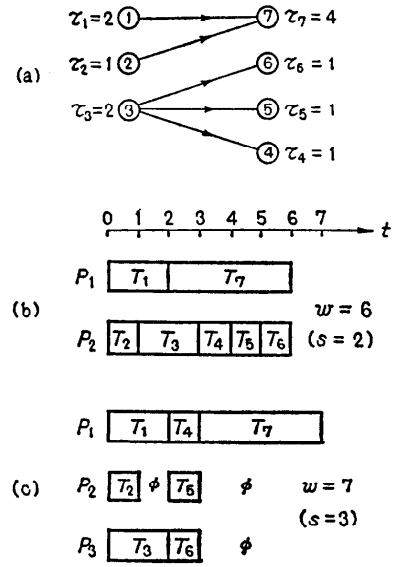


図-38 決定性スケジューリングにおける変則現象

増加率の上限は

$$w'/w \leqq 1+(n-1)/n' \qquad (6.1)$$

によって与えられる[66]. 特に $n=n'$ の場合は

$$w'/w \leqq 2-1/n \qquad (6.2)$$

が成立する. この式は「どのようなスケジュールを用いても最適のスケジュールの2倍悪くなることはない」という重要な事実を含んでいる.

第2の警告は「多くの場合, 最適スケジューリングを求めるのは困難である」ということである. 実際, 次のような単純な問題が NP 完全 (5.4参照) であることが知られている[67].

i) プロセッサの数 $s=2$, すべてのタスクが独立 ($G=(T, E)$ において $E=\phi$) の場合に τ_i; $i=1\sim n$ として任意の正整数を与えて最適スケジュールを求める問題.

ii) $\tau_i=1$; $i=1\sim n$ の場合に, 半順序関係 $G=(T, E)$; $|T|=n$ とプロセッサの数 $s\geqq 2$ を任意に与えて最適スケジュールを求める問題.

6.1.2 最適スケジューリング・アルゴリズム

ここでは多項式オーダの最適スケジューリング・アルゴリズムが存在するような特殊な場合について解説する. 以後, 特にことわらない限り $\tau_i=1(i=1\sim n)$ とする.

① $G=(T, E)$ が**有向木** (rooted tree) の場合には, 最長径路アルゴリズムによって, 任意のプロセッサ数 $s\geqq 2$ に対して最適スケジューリングが得られる[68].

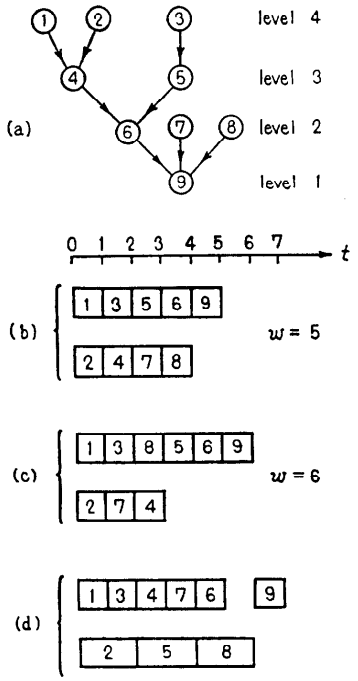


図-39 有向木に対するスケジューリング

これを**図-39** の例に沿って説明しよう.

図-39(a) のグラフは節点 (タスク) T_9 を**根**(root) とする有向木である. 根となる節点の**レベル**を1とし, 以下図-39(a) に示すように各節点のレベルを定める. このような意味のレベルは各枝の長さを1として, 根 T_9 を基準とした最長径路アルゴリズムによって容易に求められる. そしてレベル値の大きい順に高い優先順位を与えれば $(T_1, T_2, \cdots, T_9)$ というプライオリティ・リストが得られ, これに基づいて**図-39**(b) に示す最適スケジュールが得られる. これに対して, 勝手なトポロジカルソートを行うと最適スケジュールが得られるとは限らない. 例えば, 横型探索に基づくトポロジカルソートは $(T_1, T_2, T_3, T_7, T_8, T_4, T_5, T_6, T_9)$ というプライオリティ・リストを与え, **図-39**(c) に示すようなスケジュールをしてしまう.

Hu[68] は節点のレベル値に基づくラベリングが最適解を与え, その時のジョブ完了時間が

$$w=\max_{0\leqq j\leqq L}\{j+\lceil n_{j+1}/s\rceil\} \qquad (6.3)$$

で与えられることを示した. ここで L はレベル値の最大値, n_j はレベル値が j 以上の節点の数である. また $\lceil x\rceil$ は x 以上の最小の整数を表わすものとする. Hu のラベリング法はプロセッサの処理速度が異なる

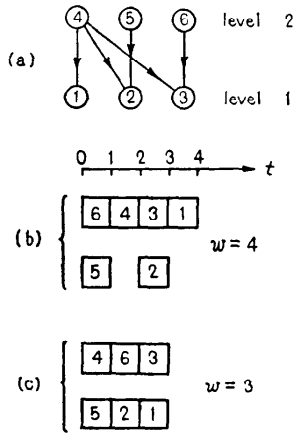


図-40　プロセッサが2個の場合のスケジューリング

場合に対してもある程度拡張できる．例えば速度比2:1の2個のプロセッサを持つ場合でも，**図-39**(d)に示すような最適解を与える．速度比3:1の場合でも多項式オーダで最適解を求められるが，特殊ケースに対する扱いが複雑となる．一方，Huのラベリングをそのまま用いても，ジョブ完了時間が，最適解より高々1延びるだけである[69]．

② プロセッサの数 $s=2$ の場合には，Huのラベリングを少し改良することによって最適解が得られる[70]．例えば**図-40**(a)のグラフに対してHuのラベリング法を適用すると $(T_6, T_5, T_4, T_3, T_2, T_1)$ というプライオリティ・リストを出力する場合があり，この場合**図-40**(b)に示すような最適でないスケジュールを行う．ところが「2個の節点(タスク) T_i, T_j の対において，T_i に直接後続する節点の集合が T_j に直接後続する節点の集合の真部分集合となる場合には T_j の優先順位を T_i のそれよりも高くする」という制限付きでHuのラベリング法を適用すれば $(T_4, T_5, T_6, T_3, T_2, T_1)$ というプライオリティ・リストが得られ，**図-40**(c)に示すような最適スケジューリングを行う．

上記のアルゴリズムの最適性は証明されている[70]が，

- $s \geqq 3$
- $s=2$ でプロセッサの速度が異なる
- タスク処理時間 τ_i が異なる

のいずれの場合においても，多項式時間内に最適解を求めるアルゴリズムは知られていない．

③ **割り込み可能** (preemptive) スケジューリングにおいては，タスク処理時間 τ_i に関する制限を取り除いても，問題の難しさに関係しない．例えば，**6.1.1**において提起した NP 完全問題はもはや取るに足らない問題となってしまい，最適スケジュールに対するジョブ完了時間は，明らかに次式で与えられる．

$$w=\max\left\{\max \tau_i, \frac{1}{s}\sum_{i=1}^{n}\tau_i\right\} \qquad (6.4)$$

更に①，②の特殊ケースにおいて $\tau_i=1 (i=1\sim n)$ という制限を除去しても，割り込み可能ならば，ほぼ同様のアルゴリズムによって，式(6.4)の最短完了時間を実現することができる[71]．先ず，$\tau_i>1$ のタスクがあったら，それを τ_i 個のタスクの系列に分割して，すべてのタスクの処理時間が1となるようなタスクシステム $\mathbf{G}'=(T', E')$ に変換する．次に $\boldsymbol{G}'$ に対して最長径路アルゴリズムを適用して①，②におけるような節点のレベル値を求める．更に，レベルの高い順にタスクをプロセッサに割り付けるのであるが，等しいレベルの節点(タスク)の数 k が

$$k>s \quad \text{且つ} \quad k \bmod s \neq 0 \qquad (6.5)$$

を満たす場合には，遊んでいるプロセッサがないようにどれかのタスクを再分割しなければならない．

例えば，**図-41**(a)のタスク・システムは，$\tau_1=\tau_2=\tau_4=2$ であるので，**図-41**(b)のように変換される．割り込みを許さない場合には，**図-41**(c)のスケジュールが最適であるが，割り込みを許せば**図-41**(d)のスケジュールが可能となる．ここでレベル3において，タスク $T_{1.2}, T_{2.2}, T_{4.1}$ が時間間隔1.5をシェアしていることに注意されたい．

上記のアルゴリズムは①，②の特殊ケースに対して最適性を保証するのであって，一般の無サイクル有向グラフや3個以上のプロセッサがある場合には拡張で

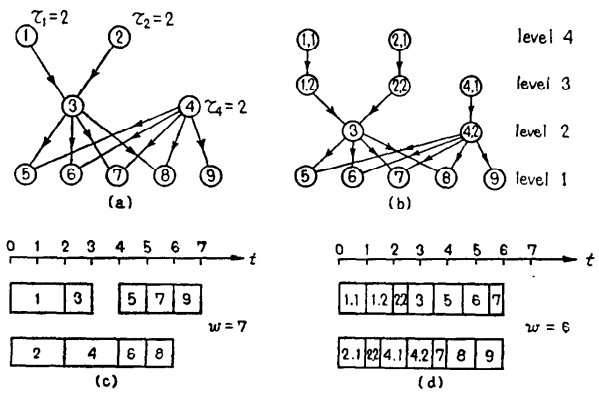


図-41　割り込み可能スケジューリング

きない.

6.1.3 スケジューリングの限界

能率の良い最適スケジューリング・アルゴリズムが存在すればOSのディスパッチャーとして使えるが,そのような場合は限定されているので,非最適スケジューリングによって管理されるコンピュータシステムの処理能力の限界を調べることも重要である.この方向の研究に対してもグラフ理論が応用されている.例えば,無サイクル有向グラフを固定したり,ラベリング・アルゴリズムを固定したりすることによって式(6.2)よりも厳密な限界が求められている.前者の方向の研究としては,文献72),73)など,後者の方向の研究としては,文献69),74)～78)などが挙げられる.

6.1.4 統計的評価

実際のOSにおいては,入力されるタスクの集合は一定してないので,ある母集団に属する入力に対する平均的な能力(TAT,スループットなど)を問題とする.すなわち,待行列理論,マルコフ過程,シミュレーション手法などを駆使した統計的評価が重要になって来る.特にシミュレーションを行う場合には,ある性質を満たすグラフを無作為に発生させる必要があり,そのための道具の一つとしてグラフ理論が利用される.例えば,Adam等は無サイクル有向グラフを無作為に発生させるための方法を提案し,これに基づいて種々のスケジューリング・アルゴリズムの良さや限界式の厳密さの統計的評価を行い,Fernandez-Busselが導いた限界式[73]が十分厳密であること,節点(タスク)の重みに基づいた最長径路アルゴリズムが平均的な意味で最も優れたスケジュールを与えることなどを主張している[79].

6.2 フローチャートの分析

プログラムの最適化を行う際の評価基準の多くは,そのグラフ的構造に関係している.すなわちフローチャート自身が有向グラフである.ここで節点はいくつかのステートメントから成る処理ブロック,枝は処理の流れを表現している.

6.2.1 ステートメントの実行頻度の測定

「プログラムのどの部分の実行頻度が高いか?」を知ることが出来れば,プログラマはその部分に工夫をこらして,プログラムを改良することができる.そこで,「前処理として,ソースプログラムの要所要所に**プローブ**(probe)を挿入し,後処理として,プローブの挿入された部分の実行頻度を測定して,プログラマに提供する」という機能を持つコンパイラが提唱されている[80].コンパイラのオーバヘッドを考慮すれば,「なるべく少ないプローブを挿入して,すべてのステートメントの実行頻度を測定したい」という要求が出て来る.以下に述べるように,この問題はグラフ理論的に解決されている.

プログラムのフローチャートは有向グラフ $G=(X, E)$ によって表現される.ある枝 $(x_i, x_j)\in E$ が同時に,節点(ブロック) x_i から出ている唯一の枝および節点 x_j に入って来る唯一の枝であれば,明らかに x_i と x_j の実行頻度が等しいので,その枝を短絡除去(**2.1**参照)することができる.すなわち,FORTRANプログラムについていえば,IF, DOなどの制御文および,レーベル付ステートメントだけが重要になって来る.また G はフローチャートを表わしているので唯一の入口($x_1\in X$ とする)と出口($x_n\in X$ とする)の対を持たなければならない.便宜上,仮想的な枝 (x_n, x_1) を付加することにする.このような処理を処した結果の有向グラフを新たに $G=(X, E)$; $|X|=n$, $|E|=m$ とおき,各々の節点(ブロック) $x_i\in X$ の実行頻度を F_i,各々の枝 $e_{ij}=(x_i, x_j)\in E$ の通過回数を f_{ij} で表わすことにする.

上記のネットワークの性質として,第1に,G は**強連結**グラフ(**2.2**参照)である.さもなければフローチャートとして意味がないからである.第2に,各々の枝の重み f_{ij} の組はグラフ G 上の**フロー**(**4.1**参照)である.すなわちフロー公理(キルヒホフの電流則)

$$\sum_{i:\, e_{ji}\in E} f_{ji}=F_i=\sum_{k:\, e_{ik}\in E} f_{ik} \qquad (6.6)$$

が成立する.この式から,すべての f_{ij}(およびすべての F_i)を測定するために必要にして十分なプローブの数は

$$\mu=m-n+1 \qquad (6.7)$$

である[81].何となればフローはグラフ G のタイセット空間に属し,その次元は G の零度 μ に等しい(**3.2**参照)からである.

測定したい量は f_{ij} ではなく F_i であるので,式(6.7)は直接役に立たない.そこで,$F_i(i=1\sim n)$ だけを測定するためのモデル[82]を導入する過程を例に沿って説明する.**図-42**(a)の強連結グラフ G の各々の節点において式(6.6)が成立するので,各節点を入口○と出口●に分割して**図-42**(b)に示すようなグラフ $\hat{G}$ を得る.$\hat{G}$ においては,G の節点の重み F_i は枝の重みに変換され,f_{ij} および F_i の組は実線の枝のフローになっていることに注意されたい.各々のカットセットに対して成立するフロー公理の組において変数

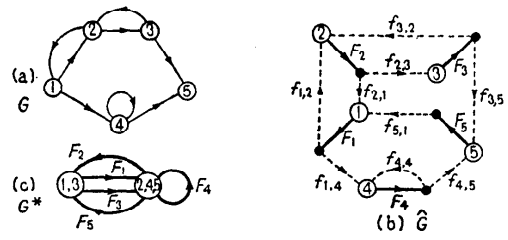


図-42　プローブ挿入の最適化

f_{ij} を消去して，F_i の間だけの関係を残すためには，図-42（b）の破線の枝をすべて短絡除去してしまえば良い．結果として**図-42**（c）のグラフ G^* が得られる．G^* に対して式 (6.7) を適用すれば，

$$\mu^*=5-2+1=4$$

個所にプローブを挿入するのが最適であることがわかる．プローブの挿入箇所は G の節点 x_4 および x_1, x_2, x_3, x_5 の中の任意の3個の節点である．この手順の最適性はタイセット空間の原始基底（**3.3** 参照）の概念から導かれる．

6.2.2　プログラムのセグメンテーション

プログラムをいくつかの"殆んど"独立なモジュールに分割するという**セグメンテーション**(segmentation)の問題は中，大型計算機のメモリ階層を管理する際に発生する問題である．仮想記憶の機能を持たないとすれば，プログラムの大きさが主記憶の容量を越えている場合，これを実行以前にいくつかのセグメントに分割し，セグメント間の制御のためのオーバレイ構造を作っておかなければならない．各セグメントを節点に対応させれば，オーバレイ構造は1つの有向木であるとみなされる．

前と同じように，プログラムは唯一の入口と出口の対を持つ有向グラフによって表現されているとする．各節点の重みは，対応するモジュールのメモリ占有量を表わすものとする．プログラム全体の制御を能率良く行うためには，共通の有向閉路に属する節点（モジュール）は同じセグメントに属していることが望ましい．そのようなオーバレイ構造を得るには，与えられた有向グラフ G を強連結成分に分解し，同じ成分に属する節点を1つの節点に縮約することによって無サイクル有向グラフ G' を導けば良い[83]，G' における節点の重みは縮約された G の節点の重みの和である．G' において，入口から出口に至る最長径路の長さが主記憶の容量を越えていなければ，直ちにオーバレイ構造が得られる．すなわち入口を基準とした**最長径路木**（**4.4** 参照）を求め，それに属さない枝の先端をその終点から取り外してしまえば，入口を根とする有向木が得られ，それがオーバレイ構造を表わす．

最長径路木の長さが主記憶容量を越えていれば，プログラムの制御は複雑になるが，より細かい分割をしなければならない．一般的な問題として，プログラムを等しいサイズのいくつかのページに分割する問題を考えよう．ここで目的関数は異なるページ間の遷移の回数の総和を最小にすることである．この組み合せ計画問題は"難しい"問題のクラスに属すると考えられ，実用的にはヒューリスティック算法に頼らなければならない[84]．ただし次に述べるような制限付き分割問題に対しては，能率良い最適分割のアルゴリズムが存在する．

有向グラフ $G=(X, E)$ の各節点が1から $n=|X|$ まで番号付けされていて，同じ分割成分に属する節点の番号は連続していなければならないとする．この番号付けの意味としてはコーディングシートに表われる順番を想定すれば良い．更に前と同じように，各分割成分に属する節点の重みの和は与えられたページサイズ p 以下という制限も加わる．目的関数は異なる分割成分を結ぶ枝の重み（遷移の回数）の総和を最小にすることである．Kernighan はこの問題に対して，動的計画法を適用して最適解を $0(m)$: $m=|E|$ の 時間内に求めるアルゴリズムを導いた[85]．**図-43** の例について結果だけを述べる．

与えられたグラフ（フローチャート）を**図-43**（a）に示す．ページサイズは $p=2$，節点の重みはすべて1

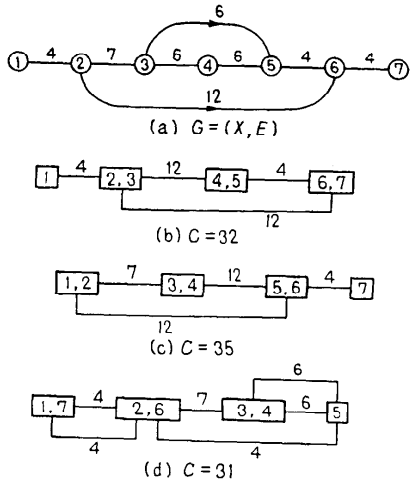


図-43　グラフの連続分割

とする．また図における無向枝は両方向の枝が存在することを示し，2つの枝の重みの和を無向枝の重みとして与えてある．これに対して Kernighan の方法を適用すれば，**図-43**（b）に示す最適分割が得られ，総コスト $C=32$ となる．通常のコンパイラは**図-43**（c）のような分割をすることが多く，そのときのコストは $C=35$ となる．一方，節点番号の連続性を無視すれば，**図-43**（d）の分割が最適となり，コストは $C=31$ となる．

6.2.3 コードの改善

ソースプログラムから冗長な計算を取り除いたり，実行頻度の高い部分に置かれているステートメントを実行頻度の低い部分に移すことによってオブジェクトコードが改善されることがあり，このような改善の対象となる部分を抽出する際に，有向グラフの構造が本質的な役割を演じる[86),87)]．

唯一の入口 x_1 と出口 x_n を持つ有向グラフ $G=(X, E)$ の節点対 (x_i, x_j) において，x_1 から x_j に至るすべての有向道が x_i を通るとき，x_i は x_j を**支配する**（dominate）という．例えば**図-44**（c）のグラフにおいて x_1 が入口であり，x_2 は x_3, x_4, x_5, x_6 のすべては支配している．この“支配”の概念を用いて，除去可能なステートメントを抽出したり[86)]，反復ループの中から外に引き出せるステートメントを抽出したりすることができる[87)]．ここでは後者について，例を挙げて説明する．

グラフ G の有向閉路 C が唯一の入口を持ち，その中のあるステートメントに関連した変数の値が C の中で不変であるならば，そのステートメントを C の外に引き出すことができる．そして，節点 x_i が C の唯一の入口であるための必要十分条件は，$(x_i, x_i)\in E$ または $(x_j, x_i)\in E$ かつ x_i が x_j を支配するような節点 x_j が存在することである[87)]．例えば，**図-44**（a）の FORTRAN プログラムは**図-44**（b）のプログラムと等価であり，且つ後者の方がはるかに優れている．このプログラムの変換は有向グラフにおける図-44（c）から**図-44**（d）への変換に対応している．ここで節点 x_4 が内側のループの唯一の入口，節点 x_3 が真中のループの唯一の入口であるという性質が利用されている．

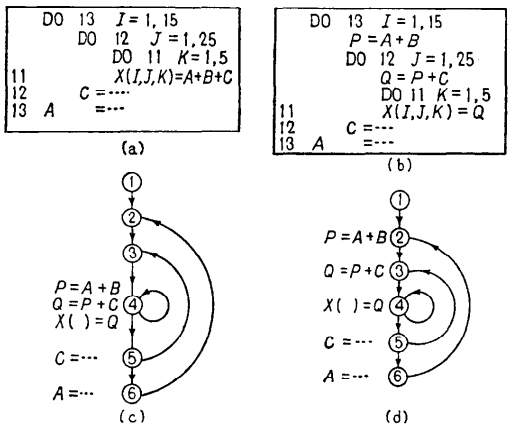


図-44　コードの改善

6.3 並列処理機能のモデル化（ペトリネット）

並列処理機能を持つシステムの有向グラフによる表現やその解析法についてはある程度論じて来た．並列な動作を表現するだけなら単なる有向グラフで十分であるが，「どのような条件に基づいて並列動作の**同期**（synchronization）を取るか？」という本質的な要因を記述するのには不十分である．同期の問題も考慮したグラフ的モデルも種々提案されている[88)]が，本講では最近最も流行している**ペトリネット**（Petri Nets）についての概略を紹介する．より詳しい解説としては，例えば文献[89)]がある．

ペトリネットを形式的に定義すると，有向2部グラフ $G=(P\cup T, I\cup O)$ と P から自然数の集合 N への写像 $S: P\to N$ の対 $PN=(G, S)$ によって表現される．ここで $P=\{P_1, P_2\cdots p_n\}$ は**場所**（place）と呼ばれ図の上では○によって表現され，$T=\{t_1, t_2,\cdots t_m\}$ は**遷移**（transition）と呼ばれ｜によって表現される．$I\subset P\times T$ の要素は場所から遷移に向う**アーク**（有向枝）――**入力枝**（input arc）と呼ばれる――，$O\subset T\times P$ の要素は遷移から場所に向う**アーク**――**出力枝**（output arc）と呼ばれる――である．また $S=(s_1, s_2, \cdots, s_n)$ は**マーク付け**（marking）あるいは**状態**（state）と呼ばれ $s_i(i=1\sim n)$ の値を p_i の○の中に描いた●――**トークン**（token）と呼ばれる――の数によって表記される．**図-45** はペトリネットの例である．

遷移 t_j は，その入力場所 $p_i: (p_i, t_j)\in I$ のすべてにおいて少なくとも1個のトークンがあるとき，**発火可能**（firable）であるという．遷移 t_j が発火すれば各各の入力場所 p_i $(p_i, t_j)\in I$ から1個ずつトークンが除去され，各々の出力場所 $p_k: (t_j, p_k)\in O$ に1個ずつトークンが加えられ，結果として新しい状態が得られる．発火可能な遷移が複数個あったら，その中の1個が無作為に選ばれて発火するという約束がもうけられている．遷移が1個ずつ発火して状態が変化して行

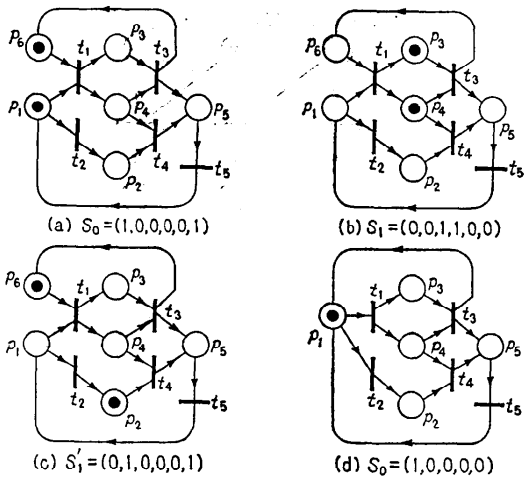


図-45　ペトリネットの例

く過程を**実行系列** (execution sequence) という.

ペトリネットにおける遷移はシステムの中の事象を，場所はそれが生起するための条件を抽象化したものである．すなわちある遷移の入力場所のすべてがトークンを持っていることは，対応する事象が生起するための条件を満たしていることを示す．また，実行系列はモデル化の対象となっているシステムのシミュレーションに相当する.

各々の場所が，如何なる実行系列においても高々1個のトークンしか持ち得ないならば，ペトリネットは**安全** (safe) であるといわれる．高々 $K(\geq 2)$ 個のトークンしか持ち得ないならば ***k*—安全** (k—safe)，高々有限個のトークンしか持ち得ないならば，**有界** (bounded) であるといわれる．状態 S から出発して状態 S' に至る実行系列が存在するとき，S は S' に**到達可能** (reachable) であるという．状態 S から到達可能な状態の集合を $r(S)$ で表わす．遷移 t_i と状態 S に対して，各々の状態 $S'\in r(S)$ から出発して t_i が発火可能な状態に到達可能ならば，t_i は S に関して**生きている** (live) といい，そうでなければ，**死んでいる** (dead) という．すべての遷移が生きているならばペトリネットは**生きている**といわれる*．発火可能な複数個の遷移が共通の入力場所を共有しているとき，それらは**競合している** (in conflict) という.

例えば**図-45** (a) のペトリネットにおいて，t_1 と t_2 の両方が発火可能であるが，場所 p_1 を共有しているので，t_1 と t_2 は互いに競合している．**図-45** (b), (c) はそれぞれ t_1, t_2 が発火した結果の状態を示している．図(c)の状態では，どの遷移も発火できないのでこれらのペトリネットは死んでいる．また．どの場所もトークンを高々1個しか持ち得ないので，安全である．一方，このペトリネットを一部変形した**図-45** (d) のペトリネットでは p_1 のトークンの数が無限に増大する可能性があるので，有界でない.

システムをペトリネットでモデル化することによる効果の1つは，システムが内蔵する**並行性** (concurrency) とそれに伴う(クロックによるのではなく，事象の生起のための条件が満たされているかによる) 同期のメカニズムを記述できるということである．例えば“競合”という概念は同期に対する制約を表わしている．更に，トークンの移動という概念によって，システムの持つ動的な側面を記述できることや，遷移の発火における無作為性によってペトリネットが**非決定性** (nondeterministic) であることが単なるグラフ(ネットワーク) と異なる点である．ペトリネットのもう1つの利用法は，それを解析(シミュレーション) することによって，システムが望ましい(あるいは望ましくない) 性質を持っているか否かを調べることである．ちなみに“生死”は**デッドロック** (deadlock) の有無に関連し，“安全”，“有界”はリソース(の容量) の制限に関連している.

ペトリネットによるモデル化の簡単な例を2,3挙げよう．**図-46** は並列処理プログラムの記述の例である．図において，処理ブロック B1.3 は B1.1 と B1.2

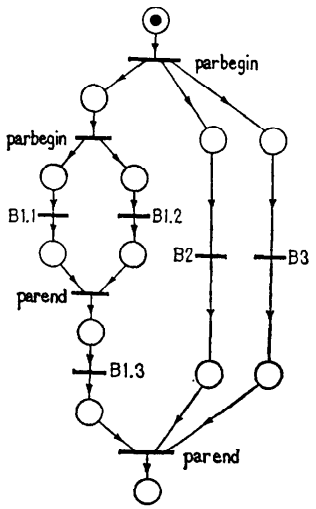


図-46　並列計算のモデル化

* 安全(有界)性，生死に関しては目的に応じて種々の定義がある[9].

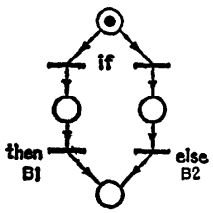


図-47 分岐のモデル化

の両方が終了しなければ開始できないこと，B1.3, B2, B3 のすべてが終了しなければ次の処理に移れないという同期の制約が表現されている．一方図-47 は普通の（並列処理を前提としない）プログラムにおける分岐を記述しており，B1 と B2 の内，片方しか実行出来ないことが表現されている．このようにペトリネットによれば，並行処理できる場合とできない場合が明確に区別される．

ペトリネットはソフトウェアだけでなくハードウェアの記述にも有効である．例えば，CPU の中のアキュムレータ，メモリデータ，アドレスレジスタ，インストラクションレジスタ，ロケーションカウンタ，などの構成要素間の動的な因果関係を明確に記述できる[90]．

図-48 は OS におけるデッドロックの問題に関連したものである．3つのリソース（例えば，カードリーダ，ラインプリンタ，テープドライブ）があり，3つのプロセス A, B, C の各々が，その内2つのリソースを使うとする．もし各々のプロセスが1個のリソースを保持しながら，もう1つが空くのを待ち続ければ，デッドロックとなる．デッドロックを避けるために，唯一のプロセスしかリソースをアクセスできないという同期の取り方に対する制約を記述したのが図-48 である．

ペトリネットは事象の競合関係を表現できるが，事象の生起に対する優先順位を表現できない[91]．この方向への拡張として，**禁止枝** (inhibitor) と呼ばれる特別な入力枝が定義され，「遷移の通常の入力枝の始点となる場所のすべてがトークンを持ち，且つ禁止枝の始点となる場所が空である場合に発火可能」という規則を設定することが提案されている[92]．これによって優先順位の表現は可能になるが，ペトリネットの解析の方はより困難になる．

ペトリネットのモデル化能力を高めるためにそれを拡張するという方向の研究も多いが，逆に解析能力を高めるためにペトリネットの部分集合に議論を限定す

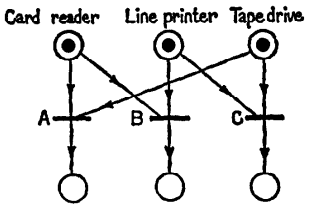


図-48 リソース利用における競合関係

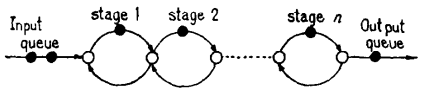


図-49 パイプライン演算装置のモデル化

るという方向の研究もある．その代表的なものが**マーク付きグラフ** (marked graph)[93] という概念である．ペトリネットにおいて，各場所が唯一の入力枝と出力枝の対を持てば，それを1個の枝で表現することによって，遷移を表わす節点と場所を表わす枝の組から成る有向グラフに帰着することができる．そして，トークンの数だけ各枝の上に●を付けたものがマーク付きグラフである．図-49 は行列計算の高速化に有効なパイプライン演算装置をマーク付きグラフで表現——○は遷移を表わす——したものである．厳密にいえば，もっと複雑なマーク付きグラフが必要であるが，図-49 だけでも，「入力キューから入って来て出力キューに送り出される定状的なデータの流れがある限り，各段階の計算は並行に実行される」という原理が説明されている．

一般のペトリネットに関しては，有効な解析手段が殆んど知られていないが，マーク付きグラフに関してはグラフ的アルゴリズムが威力を発揮している．また，次のような重要な結果も得られている[93]．

① 生きているための必要十分条件は，各閉路に沿って少なくとも1個のトークンがあることである．

② 安全であるための必要十分条件は，すべての枝（場所）がトークンの総和がちょうど1の閉路に含まれることである．

ペトリネットにおいては，モデル化の能力を高めるために概念を拡張すれば，それの解析が困難になり，逆に解析能力を高めるために範囲を限定すれば，モデル化能力が低下するという傾向がある．このモデル化能力と解析能力の間の二律背反的な関係が今後の研究の方向付けにおける大きな要因となろう．

7. あとがき

"最新の成果を盛り込みながら，且つ初心者にも読み易く"ということを狙ってグラフ理論の解説を試みたが，意図に反して中途半端な部分も多かったと思われる．しかし，情報処理という立場からグラフ理論をながめるという意味で，一応の工夫はしてみたつもりである．筆者の非力のため，不正確な記述や重要な事実の見落しなど多々あると思われるが，本稿が読者のために少しでもお役に立てれば幸である．

本稿の記述（特に(1)，(2)）において，東大伊理正夫教授のお書きになった論文や解説を参考にさせていただいた部分がいくつかある．紙面において感謝の意を表わす次第である．

参考文献

(既出分への追加)

59) 渡部 和：線形回路理論，昭晃堂 (1971).

60) Tewarson, R.: Sparse Matrices, Academic Press, New York (1973).

61) 大附辰夫，川北建次：スパース行列処理技法(1)～(3)，情報処理，Vol. 17, 1～3 (1976).

62) 可児賢二，大附辰夫：設計自動化におけるグラフ理論と組み合わせ算法(1)～(3)，情報処理，Vol. 16, 5～7 (1975).

63) Conway, R. et al.: Theory of Scheduling, Addison-Wesley (1967).

64) Coffman, E. and Denning, P.: Operating System Theory, Prentice-Hall (1973).

65) Coffman, E. (ed.): Computer and Job-Shop Scheduling Theory, John Wiley (1976).

66) Graham, R.: Bounds on Multiprocessing Anomalies and Packing Algorithms, Proc. AFIPS 1972 SJCC, Vol. 40, pp. 205-17.

67) Ullman, J.: Polinomial Complete Scheduling Problems, Proc. 4th Symp. on Operating Systems Principles, pp. 96-101 (1973).

68) Hu, T. C.: Parallel Sequencing and Assembly Line Problems, Operations Research, Vol. 9, No. 6, pp. 841-48 (1961).

69) Baer, J.: Optimal Scheduling on Two Processors of Different Speeds in **Computer Architectures and Networks**, Gelenbe, E. and Mahl, R. (eds.), pp. 27-45, North Holland (1974).

70) Coffman, E. and Graham, R.: Optimal Scheduling for Two Processor Systems, Acta Informatica, Vol. 1, No. 3, pp. 200-213 (1972).

71) Muntz, R. and Coffman, E.: Optimal Preemptive Scheduling on Two-Processor Systems, IEEE-TC, Vol. 18, No. 11, pp. 1014-1020 (1969).

72) Ramamoorthy, C. et al.: Optimal Scheduling Strategies in a Multiprocessor System, IEEE-TC, Vol. 21, No. 2, pp. 137-146 (1972).

73) Fernandez, E. and Bussel, B.: Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules, IEEE-TC, Vol. 22, No. 8, pp. 745-751 (1973).

74) Chen, N. F. and Liu, C. L.: On a Class of Scheduling Algorithms for Multiprocessors Computing Systems, **Parallel Processing, Lecture Notes in Computer Science**, Vol. 24, pp. 1-16 (1974).

75) Chen, N. F.: An Analysis of Scheduling Algorithms in Multiprocessor Computing Systems, IU-CS: R-75.0724, University of Illinois (1975).

76) Lam, S. and Sethi, R.: A Bound on Coffman-Graham Schedules for Three or More Processor Systems, TR-156, Computer Sciensce Dept., Penn. State Univ. (1974).

77) Kohler, W.: A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems, IEEE-TC, Vol. 24, No. 12, pp. 1235-1238 (1975).

78) Kaufman, M.: An Almost-Optimal Algorithm for the Assembly Line Scheduling Problems, IEEE-TC, Vol. 23, No. 11, pp. 1169-1174 (1974).

79) Adam, T. et al.: A Comparison of List Schedules for Parallel Processing Systems, C. ACM, Vol. 17, No. 12, pp. 685-690 (1974).

80) Russel, E. and Estrin, G.: Measurement Based Automatic Analysis of FORTRAN Programs, Proc. AFIPS 1969 SJCC, Vol. 34, pp. 723-732 (1969).

81) Knuth, D.: The Art of Computer Programming: Seminumerical Algorithms, Vol. 2, Addison-Wesley (1969).

82) Knuth, D. and Stevenson, F.: Optimal Measurement Points for Program Frequency Counts. BIT, Vol. 13, pp. 313-322 (1973).

83) Ramamoorthy, C.: Analysis of Graph by Connectirity Considerations, J. ACM, Vol. 13, No. 2, pp. 211-222 (1966).

84) Baer, J. and Caughey, R.: Segmentation and Optimization of Programs from Cyclic Structure Analysis, Proc. AFIPS 1972 SJCC, Vol. 40 pp. 23-36 (1972).

85) Kernighan, B.: Optimal Sequential Partitions of Graphs, J. ACM, Vol. 18, No. 1, pp. 34-40 (1971).

86) Lowry, E. and Medlock, C.: Object Code Optimization, CACM, Vol. 12, No. 1, pp. 13-22 (1969).

87) Aho, A. and Ullman, J.: The Theory of Parsing, Translation and Compiling, Vol. II, Prentice-Hall (1973).

88) Baer, J.: A Survey of Some Theoretical Aspects of Multiprocessing, Comp. Surveys, Vol. 5, No. 1, pp. 31-80 (1973).

89) Peterson, J.: Petri Nets, Computing Surveys, Vol. 9, No. 3, pp. 223-252 (1975).

90) Patil, S. and Dennis, J.: The Description and Realization of Digital Systems, COMPCON 72 Digest, pp. 223-227 (1972).

91) Kosaraju, S.: Limitation of Dijkstra's Semaphore Primitives and Petri Nets, Proc. 4th Symp. on Operating Systems Principles, pp. 122-126 (1973).

92) Agerwala, T. and Flynn, M.: Comments on Capabilities Limitations and 'Correctness' of Petri Nets, Proc. 1st Symp. on Computer Architecture, pp. 81-86 (1973).

93) Commoner, F. et al.: Marked Directed Graphs, J. Computer and System Sciences, Vol. 5, pp. 511-523 (1971).

(昭和55年5月30日受付)