プログラミング演習

埼玉大学工学部電気電子物理工学科

伊藤研究室

プログラミング演習課題

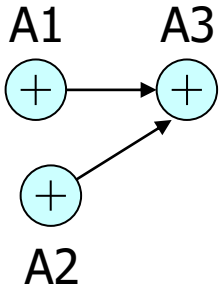
- リストスケジューリング(List Scheduling)を実行するプログラムをC/C++言語で作成する

概要

1. リストスケジューリングとは
2. データ構造
3. ファイルからデータ読み取り
4. 演算(ノード)の優先度を求める
5. 演算の実行開始時刻を決める

スケジューリングとは

- 演算(加算、乗算) の正しい実行時刻を定めること
- 前提条件: 演算間に順序関係(先行制約関係)が指定される場合はそれを守らなければならない
  - 先行制約を与える代表的なものは演算間のデータ依存関係
  - データ依存関係が存在すると、データ生成演算とデータ消費演算の間の実行順序を守る必要あり
    (順序に反すると正しくデータ授受されない)



A3は、A1とA2の演算結果を使用する

A1とA2が演算実行完了後にA3を実行

スケジューリングとは

- 先行制約をすべて満たすような演算実行時刻の組み合わせは一般に複数通り存在
- ある制約の下である評価が最適なスケジュール(解)を求める「組み合わせ最適化問題」
- 資源制約スケジューリング
  - 制約: 同時に実行する演算数が指定演算器数以下
  - 評価: すべての演算完了が最も早い
- 時間制約スケジューリング
  - 制約: 指定された時間内で全演算を実行完了させる
  - 評価: 必要な演算器数が最少

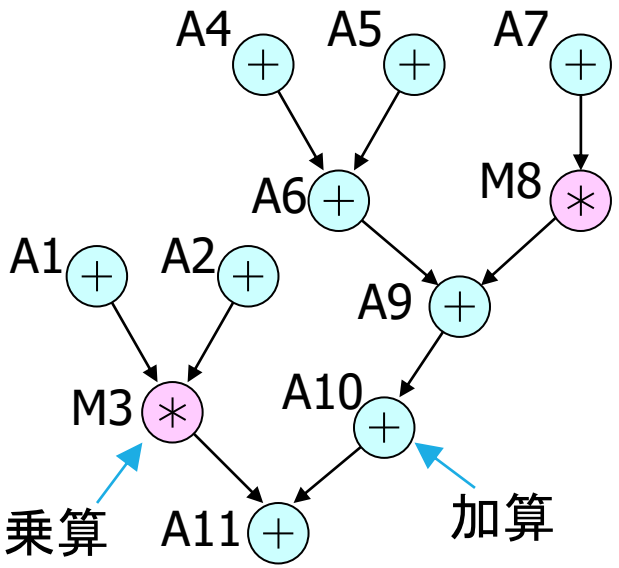
(時間制約スケジューリングは制約が不適切だと制約を満たす解が存在しない場合あり)

リストスケジューリングとは

- スケジューリングは
  最適解を求めるのが「難しい」問題(NP困難)
- 解の最適性を保証しないが、短時間で良好な(準最適な)解を求めるアルゴリズムを使用
- その1つがList Scheduling(リストスケジューリング)アルゴリズム
- 資源制約スケジューリングのアルゴリズムの1つ
- 演算器数を指定して、すべての演算の実行完了までの時間を最短化する

リストスケジューリングの効果

- 演算実行時間は加算は1、乗算は2と仮定
- 演算器数は加算器は2個、乗算器は1個と仮定



演算番号順に実行時刻を求めた場合

時刻			
0	A1	A2	
1	A4	A5	M3
2	A6	A7	
3			M8
4			
5	A9		
6	A10		
7	A11		

リストスケジューリングを適用した場合

0	A7	A4	
1	A5	A1	M8
2	A2	A6	
3	A9		M3
4	A10		
5	A11		

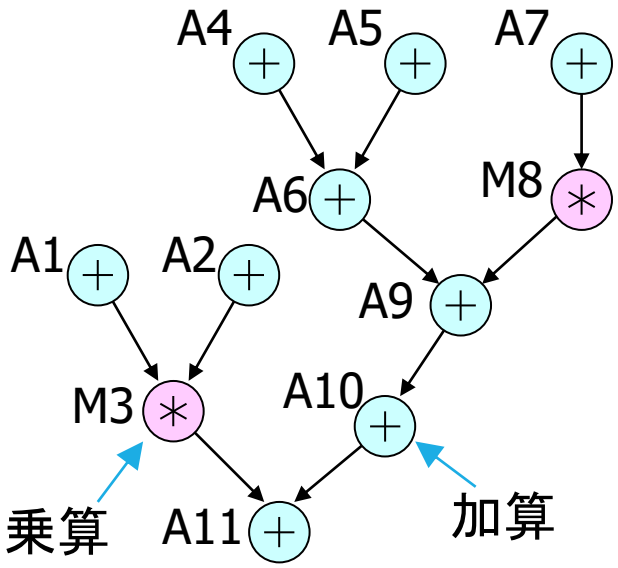
リストスケジューリングにより、
実行時間の短いスケジュールが得られる

リストスケジューリングの方法

- 「先に実行すべき演算は先に実行する」
- 1. 演算に優先度を定め、優先度の高い演算から順に実行時刻を決定する
- 2. 優先度順に選んだ1つの演算について、先行制約と演算器数制約の下で、最も早く実行可能な時刻をその演算の実行時刻と定める
- 3. すべての演算の実行時刻を定めるまで2.を繰り返す

入力データ: データフローグラフ

- データフローグラフ(Data-Flow Graph, DFG)とは
- 演算をノード(点)、演算間のデータ依存関係を有向枝(矢印)で表したグラフ



DFGの例

A1, M3などはノード(演算)の名前
(名前の付け方に特に決まりはない)

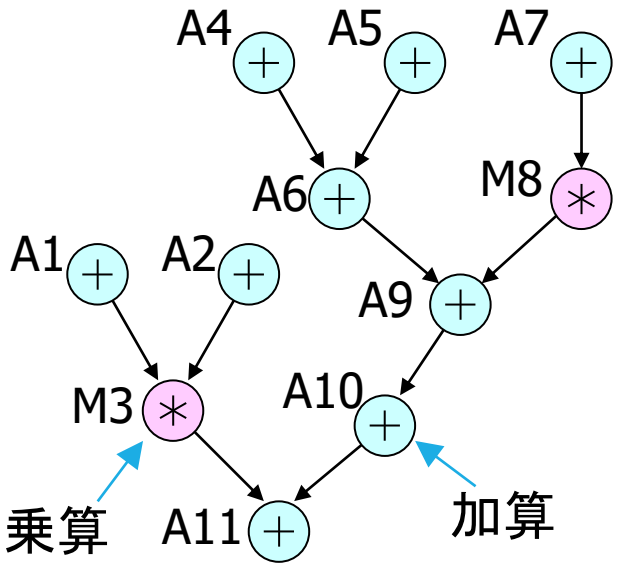
例えば、加算A1の結果が乗算M3の入力データとして使用される

外部からの入力データ、定数倍乗算の定数入力は省略している

DFGの書式

- DFGは、テキストファイルから読み込む

DFG → DFGを記述するテキストファイル




```
P A 1
P M 2
N A1 A
N A2 A
N M3 M
   :
E A1 M3
E A2 M3
E M3 A11
   :
```

DFG記述書式

- P *Ntype Time*

  演算種類を定義し、*Ntype*の演算時間が*Time*

- N *Name Ntype*

  演算ノードを記述し、名前*Name*、演算種類*Ntype*

- E *From To*

  データ依存枝を記述し、始点*From*、終点*To*

概要

1. リストスケジューリングとは
2. データ構造
3. ファイルからデータ読み取り
4. 演算(ノード)の優先度を求める
5. 演算の実行開始時刻を決める

プログラム作成に当たって

- 「アルゴリズム+データ構造=プログラム」
  (N.ヴィルト著、1975年に出版された書籍のタイトル)

- 今の場合は
- 「アルゴリズム」はリストスケジューリング
- アルゴリズムを実装するために適切な「データ構造」(データの表現方法)を用いる

演算のデータを記録する

- 演算(ノード)に関するデータ･･･構造体で表す

```
typedef struct NodeRec {
char *cName;                    // 名前(文字列)
int nNtype;                     // 演算種類(種類番号)
int nComp;                      // 実行時間
int nTime;                      // 実行開始時刻
int nPriority;                  // 優先度
struct EdgeRec *listInEdge;     // 入力枝情報 } 後述
struct EdgeRec *listOutEdge;    // 出力枝情報 }
} NODE;
```

- 全演算を表すデータ構造として構造体配列を利用

NODE node[10]; または struct NodeRec node[10];

ただし、配列要素数(ノード数)はDFGの内容に基づいて決まる

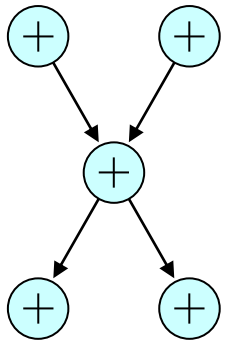
枝のデータを記録する

- 枝に関するデータ

```
typedef struct EdgeRec {
int nFrom;     // 始点ノード番号
int nTo;       // 終点ノード番号
} EDGE;
```

← 枝は構造体で記録する

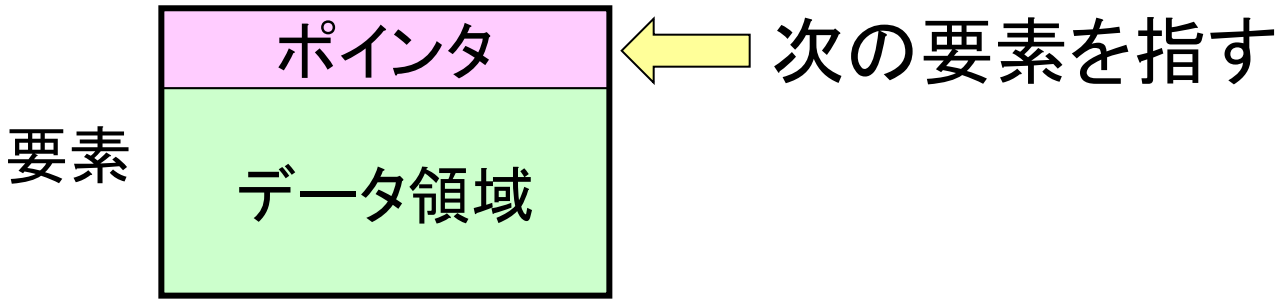
- 一般にノードには
  複数本の枝が入力、複数本の枝が出力
- 枝の数は固定ではない



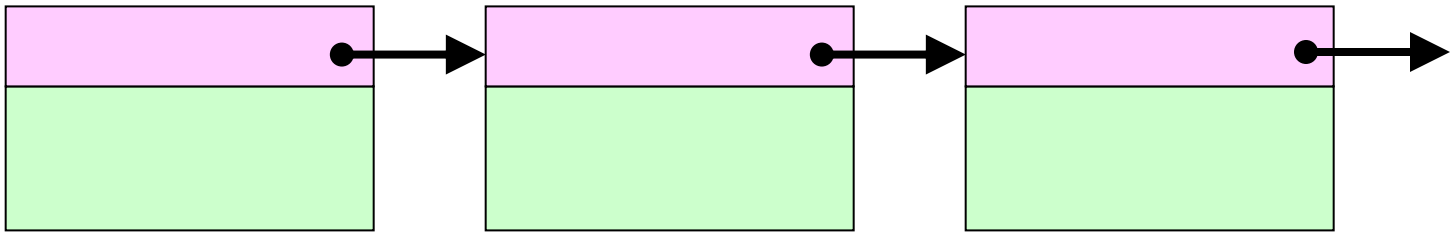
- リスト構造によって枝の接続を表す

リスト構造とは

- 要素が順につながったデータ構造(Linked list)
- データ領域の他に、次の要素を指すポインタを備える



- ポインタを用いて要素をつなげる=リスト (Linked)

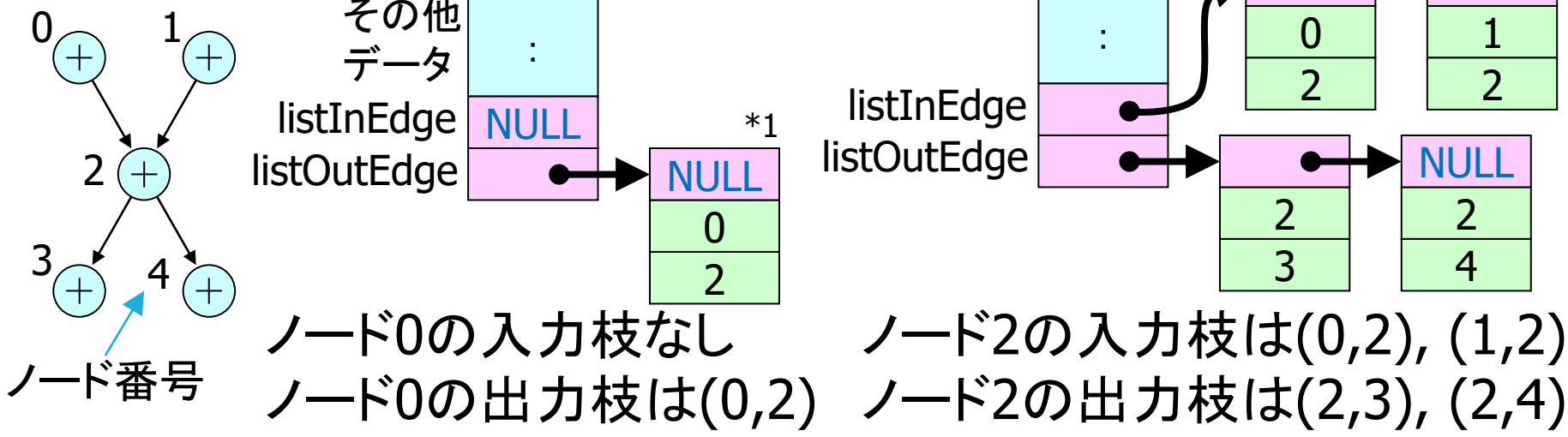


※ List schedulingの「リスト」とは直接関係なし

DFGの枝をリスト構造で表す

- リストの要素(枝)

```
typedef struct EdgeRec {
struct EdgeRec *next;   // 次の要素を指すポインタ
int nFrom;      // 始点ノード番号
int nTo;        // 終点ノード番号
} EDGE;
```



*1と*2は1つの枝(0,2)をノード2の入力枝およびノード0の出力枝として重複して2つの要素に記録している。
メモリ節約のため、1つにまとめる改善はできないか？

概要

1. リストスケジューリングとは
2. データ構造
3. ファイルからデータ読み取り
4. 演算(ノード)の優先度を求める
5. 演算の実行開始時刻を決める

ファイルからデータ読み取り

- テキストデータでファイルに記録されたDFGの情報を読み取る
- C言語プログラムからファイルを開き、内容を読み出し、解析、データ構造に記録、ファイルを閉じる

```
FILE *fp;
if( _tfopen_s( &fp, strFilename, _T("r") ) ) return;

char buf[256];
while(1){
    fgets( buf, 255, fp );
    if( feof(fp) ) break;
    // 読み込んだ行の解析とデータ記録をここで行う
}
fclose(fp);
```

ファイルを開く

ファイル名は変数strFilename
(CString型)に代入されているとする

読み出し(read)モード

ファイルが存在しないなどopenに
失敗したら処理を中止して戻る

無限ループ

1行読み込み

最後の行まで読み終わっていればループ終了

ファイルを閉じる

読み込んだ行の解析

```
char DLM[] = " ";
char buf[256];
char *ptr, *ptr1, *ptr2;
char *pTokenContext;

while(1){
    fgets( buf, 255, fp );
    if( feof(fp) ) break;
    if( strlen(buf)>0 && buf[strlen(buf)-1] == '¥n' )  buf[strlen(buf)-1] = '¥0';
    switch( buf[0] ){
    case 'P':
        ptr = strtok_s( buf, DLM, &pTokenContext );
        ptr1 = strtok_s( NULL, DLM, &pTokenContext );
        ptr2 = strtok_s( NULL, DLM, &pTokenContext );
        // 読み取ったデータを記録する処理をここで行う
        break;
    }
}
```

区切り文字(文字列を空白文字で区切る)

行末の改行文字を消す(ヌル文字に書き換え)

最初の1文字が 'P' のとき(例えば"P A 1")

"P "を読み取る

ptr1は"A"を指す

ptr2は"1"を指す

最初の1文字が 'N' や 'E' のときの処理をここに書き加える

演算種類の読み込みと記録

```
typedef struct ntypeRec {
struct ntypeRec *next;  // リストで記録
char *cNtype;       // 演算種類名
int nNtype;         // 演算種類番号
int nComp;          // 演算時間
} NTYPE;
```

```
NTYPE *pNtype, *listNtype = NULL;
int nNtypeCount=0;

switch( buf[0] ){
case 'P':
      ptr = strtok_s( buf, DLM, &pTokenContext );
      ptr1 = strtok_s( NULL, DLM, &pTokenContext );
      ptr2 = strtok_s( NULL, DLM, &pTokenContext );
      pNtype = (NTYPE *)malloc( sizeof(NTYPE) );
      pNtype->cNtype = _strdup( ptr1 );
      pNtype->nNtype = nNtypeCount++;
      pNtype->nComp = atoi( ptr2 );
      pNtype->next = listNtype;
      listNtype = pNtype;
      break;
}
```

演算種類を記録するリスト要素の記録領域を動的に確保

演算種類の文字列をコピー

演算種類に通し番号(0から開始)を付ける

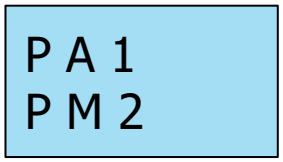
演算実行時間を表す数字文字列を数値(整数)に変換

リストに要素を加える(リスト先頭に挿入)

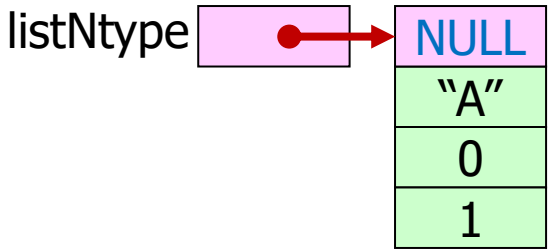
演算種類のリストの構築

- 読み込み前

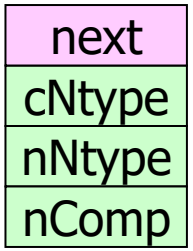
データファイル内容



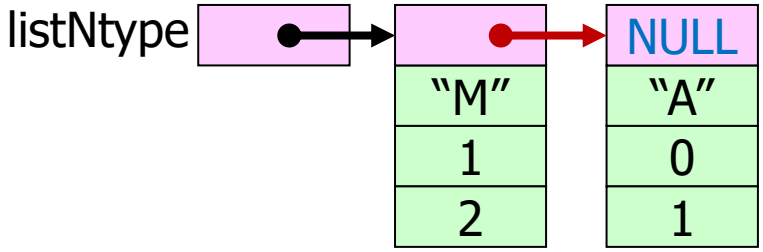
- 1行読み込み後



NTYPE



- 2行読み込み後



リストの先頭に新しい要素を挿入している

演算(ノード)の読み込みと記録

- 演算に関する情報は配列(要素は構造体)に記録
- 配列を用いるには、要素数が事前に必要

そのため

一度データファイルを全て読み込んで演算の数を調べ、配列を用意し、再度データファイルを読み込んで演算の情報を記録

```
if( _tfopen_s( &fp, strFilename, _T("r") ) ) return;
int nNodeCount=0;
while(1){
        fgets( buf, 255, fp );
        if( feof(fp) ) break;
        if( buf[0] == 'N') nNodeCount++;
}
node = (NODE *)malloc( nNodeCount*sizeof(NODE) );
fseek( fp, 0, SEEK_SET );
while(1){
        fgets( buf, 255, fp );
        if( feof(fp) ) break;
        // 読み取ったデータを記録する処理をここで行う
}
fclose(fp);
```

演算(ノード)数

ファイルの読み出し位置を先頭に移動

演算(ノード)の読み込みと記録

```
if( _tfopen_s( &fp, strFilename, _T("r") ) ) return;
int nNodeCount=0;
while(1){
	fgets( buf, 255, fp );
	if( feof(fp) ) break;
	if( buf[0] == 'N') nNodeCount++;
}
node = (NODE *)malloc( nNodeCount*sizeof(NODE) );
fseek( fp, 0, SEEK_SET );
while(1){
	fgets( buf, 255, fp );
	if( feof(fp) ) break;
	if( strlen(buf)>0 && buf[strlen(buf)-1] == '¥n' )  buf[strlen(buf)-1] = '¥0';
	switch( buf[0] ){
	case 'N':
		// 読み取ったデータを記録する処理をここで行う
		break;
	}
}
fclose(fp);
```

演算(ノード)の数を数える

演算(ノード)を記録する
構造体の配列のメモリ確保

ファイルの先頭から読み出しを再開する

ここに'N'以外の行の処理を書き加える

ファイルを閉じる

演算(ノード)の読み込みと記録

```
int nIndexNode=0;
while(1){
    // ここでファイルから1行読み込み(詳細は省略)
    switch( buf[0] ){
    case 'N':
        ptr = strtok_s( buf, DLM, &pTokenContext );
        ptr1 = strtok_s( NULL, DLM, &pTokenContext );
        ptr2 = strtok_s( NULL, DLM, &pTokenContext );
        node[nIndexNode].cName = _strdup( ptr1 );
        for(pNtype=listNtype ; pNtype ; pNtype=pNtype->next){
            if( strcmp(ptr2,pNtype->cNtype) == 0 ) break;
        }
        if( pNtype == NULL ) continue;
        node[nIndexNode].nNtype = pNtype->nNtype;
        node[nIndexNode].nComp = pNtype->nComp;
        node[nIndexNode].listInEdge = NULL;
        node[nIndexNode].listOutEdge = NULL;
        nIndexNode++;
        break;
    }
}
```

データ例

```
N A1 A
N A2 A
N M3 M
```

演算の名前を記録

演算種類名が一致するものを探す

一致するものがなければこの先の処理をスキップ*

演算種類番号、演算実行時間を記録

枝情報を記録する前の準備

(*データファイルが正しければ生じない。
演算種類名が不正ならばこの行の演算記述を無視)

枝の読み込みと記録

```
int nFrom, nTo;
EDGE *pEdge;
while(1){
    // ここでファイルから1行読み込み(詳細は省略)
    switch( buf[0] ){
    case 'E':
        ptr = strtok_s( buf, DLM, &pTokenContext );
        ptr1 = strtok_s( NULL, DLM, &pTokenContext );
        ptr2 = strtok_s( NULL, DLM, &pTokenContext );
        for(nFrom=0 ; nFrom<nNodeCount ; nFrom++){
            if( strcmp(ptr1,node[nFrom].cName) == 0 ) break;
        }
        if( nFrom >= nNodeCount ) continue;
        for(nTo=0 ; nTo<nNodeCount ; nTo++){
            if( strcmp(ptr2,node[nTo].cName) == 0 ) break;
        }
        if( nTo >= nNodeCount ) continue;
```

次ページへ続く

データ例

```
E A1 M3
E A2 M3
E M3 A11
```

枝の始点の演算名が一致するものを探す

一致するものがなければこの先の処理をスキップ*

枝の終点の演算名が一致するものを探す

一致するものがなければこの先の処理をスキップ*

(*データファイルが正しければ生じない。
始点、終点が不正ならばこの行の枝記述を無視)

枝の読み込みと記録

```
int nFrom, nTo;
EDGE *pEdge;
while(1){
    // ここでファイルから1行読み込み(詳細は省略)
    switch( buf[0] ){
    case 'E':
        // 省略
        pEdge = (EDGE *)malloc( sizeof(EDGE) );
        pEdge->nFrom = nFrom;
        pEdge->nTo = nTo;
        pEdge->next = node[nFrom].listOutEdge;
        node[nFrom].listOutEdge = pEdge;
        pEdge = (EDGE *)malloc( sizeof(EDGE) );
        pEdge->nFrom = nFrom;
        pEdge->nTo = nTo;
        pEdge->next = node[nTo].listInEdge;
        node[nTo].listInEdge = pEdge;
        break;
    }
}
```

演算nFromからの出力枝として追加

演算nToへの入力枝として追加

データ例

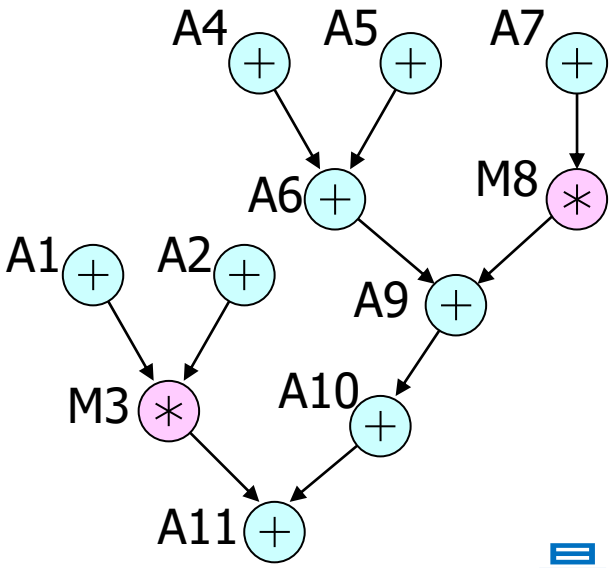
```
E A1 M3
E A2 M3
E M3 A11
```

概要

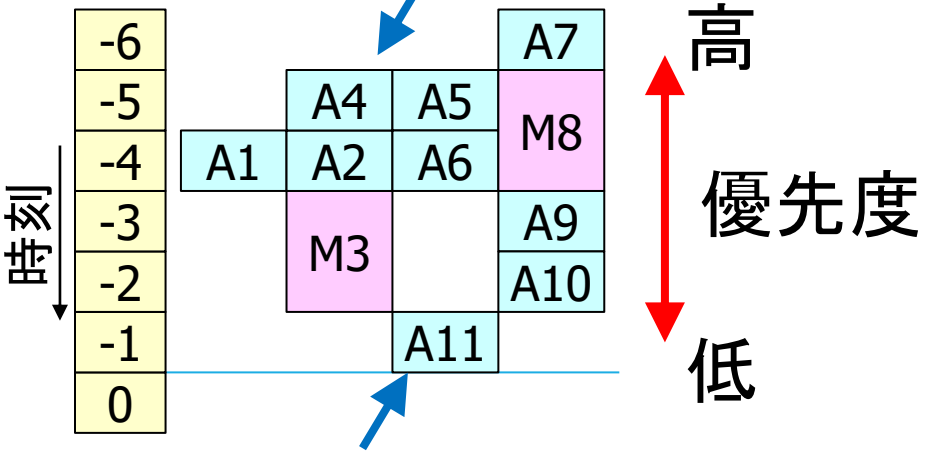
1. リストスケジューリングとは
2. データ構造
3. ファイルからデータ読み取り
4. 演算(ノード)の優先度を求める
5. 演算の実行開始時刻を決める

演算(ノード)の優先度を求める

- DFG中で最後に実行すべき演算が存在
- その演算に至る演算時間の長さ ＝ 優先度
- できるだけ遅く実行するスケジュールが優先度
  As Late As Possible, ALAP



同時実行演算数に
制約は設けない



最後の演算が時刻0に実行完了とする

演算(ノード)の優先度を求める

1. (優先度の初期値)全ての演算kについて、node[k].nPriority=-node[k].nCompとする。
2. 演算kについて以下のPを求める。

$$P = \min_{\forall(k,n)} \{\text{node[n].nPriority-node[k].nComp}\}$$

ただし、$\min_{\forall(k,n)}$ は演算kの全ての出力枝の終点nに関して最小値を求めることを意味する。

3. P <node[k].nPriorityならば、node[k].nPriorityにPを代入する。(優先度の更新)
4. 全ての演算について2.、3.を実行する。
5. 優先度の更新がなければ終了。いずれかの演算で優先度が更新されたら再度2.、3.を繰り返す。

(最長経路問題の解法の一つであるBellman-Ford法を利用)

演算(ノード)の優先度を求める

- 演算kの全ての出力枝について最小値を求める

```
int nUpdate, nPriority0;
while(1){
    nUpdate = 0;
    for(k=0 ; k<nNodeCount ; k++ ){
        nPriority0 = node[k].nPriority;
        for(pEdge=node[k].listOutEdge ; pEdge ; pEdge=pEdge->next){
            n = pEdge->nTo;
            if(nPriority0 > node[n].nPriority-node[k].nComp)
                nPriority0 = node[n].nPriority-node[k].nComp;
        }
        if(node[k].nPriority > nPriority0){
            node[k].nPriority = nPriority0;
            nUpdate = 1;
        }
    }
    if( nUpdate == 0 ) break;
}
```

全ての演算kについて

演算kの全ての出力枝について

枝の終点演算をnとする

より小さな優先度値をnPriority0に記録

優先度が更新されたことをnUpdate=1として記録

更新がなければ終了

演算(ノード)の優先度を求める

- 演算kの全ての出力枝について最小値を求める
  (簡潔な記述)

```
int nUpdate;
while(1){
    nUpdate = 0;
    for(k=0 ; k<nNodeCount ; k++ ){
        for(pEdge=node[k].listOutEdge ; pEdge ; pEdge=pEdge->next){
            n = pEdge->nTo;
            if(node[k].nPriority > node[n].nPriority-node[k].nComp){
                node[k].nPriority = node[n].nPriority-node[k].nComp;
                nUpdate = 1;
            }
        }
    }
    if( nUpdate == 0 ) break;
}
```

全ての演算kについて

演算kの全ての
出力枝について

枝の終点演算をnとする

優先度が更新されたことを
nUpdate=1として記録

更新がなければ終了

- 最高優先度(優先度の値としては最小値)を
  nPriorityMinとする

概要

1. リストスケジューリングとは
2. データ構造
3. ファイルからデータ読み取り
4. 演算(ノード)の優先度を求める
5. 演算の実行開始時刻を決める

同時に実行する演算数を記録する

- 演算種類ごとに演算器数制約を満たす必要あり
- 各時刻において同時に実行する演算数を記録
  ⇒ 二次元配列を利用

  nFUUsage[nNtypeCount][nTimeMax];

  演算種類数　　最大演算実行時刻

  nTimeMaxは、全ての演算を直列実行すると仮定した時間であり、全ての演算実行時間の和として求める
- 二次元配列を動的に確保

```
int **nFUUsage = (int **)malloc(nNtypeCount*sizeof(int *));
for(k=0 ; k<nNtypeCount ; k++){
    nFUUsage[k] = (int *)malloc(nTimeMax*sizeof(int));
    for(t=0 ; t<nTimeMax ; t++) nFUUsage[k][t] = 0;
}
```

優先度順に演算(ノード)を選ぶ

```
for(nPriority=nPriorityMin ; nPriority<0 ; nPriority++){
    for(k=0 ; k<nNodeCount ; k++){
        if(node[k].nPriority != nPriority) continue;
        // ここに来たとき、演算kは優先度nPriority
        演算kの実行時刻を決める;
    }
}
```

← 優先度の高い方から低い方へ順に調べる

演算の実行開始時刻を決める

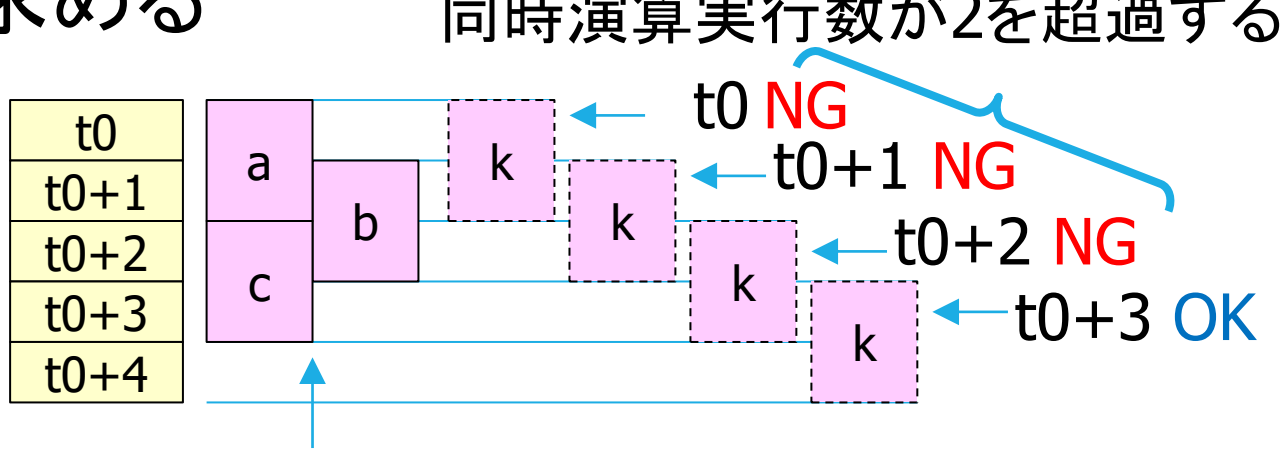
- 先行制約に基づき演算kの実行開始可能な最早時刻t0を求める

$$t0 = \max_{\forall(j,k)} \{node[j].nTime + node[j].nComp\}$$

演算kの全ての入力枝の始点jに関する最大値

- t0以降で演算器数制約を満たしてkを実行開始可能な時刻tを求める

演算器数=2
演算実行時間=2
の場合



演算の実行開始時刻を決める

```
for( t=t0 ; ; t++ ){
    for( t1=t ; t1<t+node[k].nComp ; t1++ ){
        if( nFUUsage[nNtype][t1]+1>nFUMax[nNtype] ) break;
    }
    if( t1>=t+node[k].nComp ) break;
}

node[k].nTime = t;

for( t1=t ; t1<t+node[k].nComp ; t1++ ) nFUUsage[nNtype][t1]++;
```

nFUMax[nNtype]は
演算種類nNtypeの
指定演算器数(制約)

時刻t1において同時演算実行数が
演算器数を超過する場合はforを中断

この条件が成り立つときは、上のfor文がbreakではなく正常終了した
つまり、演算を時刻tに実行開始可能であるので外側のfor文を抜ける

演算kの実行開始時刻をtに決定

各時刻における演算実行数を更新

その他

- malloc関数により確保したメモリ(データ記録領域)は、不要になったときにfree関数を用いて解放する
- 適切にメモリを解放しないと、メモリ不足となり実行低速化や異常終了の原因となる
- プログラム終了時には、そのプログラム中で確保したメモリはすべてOSによって解放されるが、きちんとプログラム側で解放するような「行儀」のよいプログラムを書く習慣を身に着ける

例:演算種類のリストの解放

- free関数を用いてメモリを解放

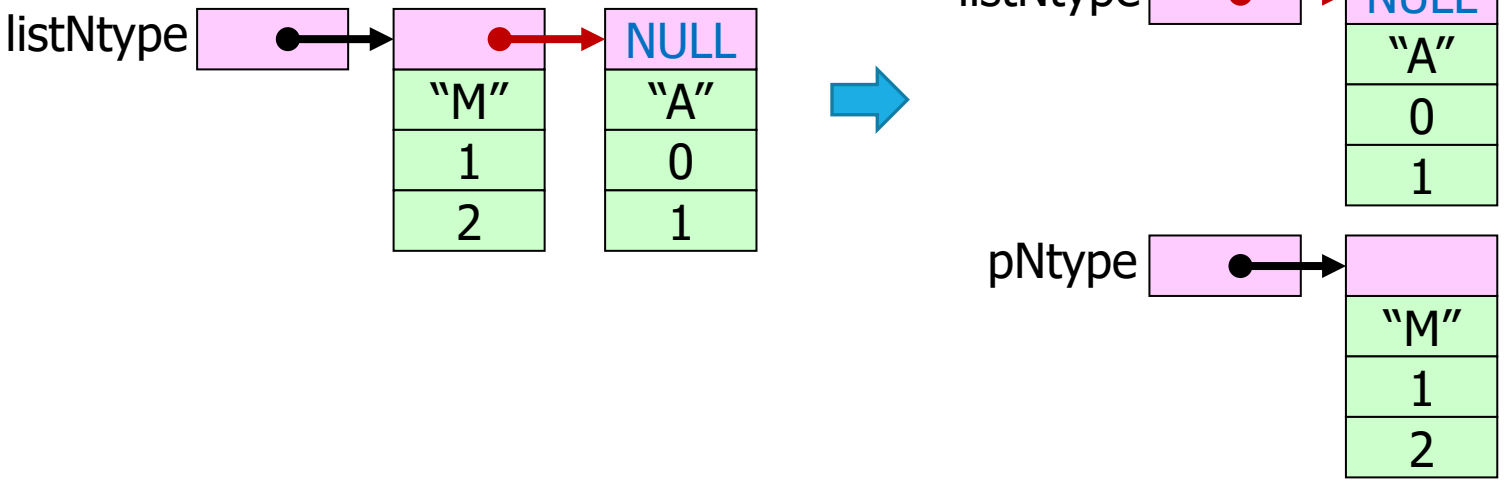
```
NTYPE *pNtype;
while( listNtype ){
        pNtype = listNtype;
        listNtype = listNtype->next;
        free( pNtype->cNtype );
        free( pNtype );
}
```

リストに要素がある間は繰り返し

リストの先頭要素をリストから外し、
その要素はpNtypeが指し示す
(下図参照)

演算種類の文字列を記録したメモリを解放

演算種類の要素を記録したメモリを解放



以上