半導体等製造装置におけるホイスト スケジューリング問題に対する発見的手法

野々部　宏司

半導体等の製造において，自動製造装置の稼働効率を高めるために装置内の搬送機（ホイスト）の動作スケジュールを最適化することが求められている．この問題はホイストスケジューリング問題と呼ばれ，no-wait 型フローショップスケジューリングに種々の条件が付加された問題と捉えることができる．本稿では，現実の半導体等製造装置に現れるホイストスケジューリング問題を対象として，ネットワークアルゴリズムと局所探索法を用いた発見的手法を提案する．なお，本稿は文献 [1, 2] の内容を加筆修正したものである．

キーワード：ホイストスケジューリング，発見的手法，局所探索法，ネットワークアルゴリズム，最長路問題

1. はじめに

本稿では，現実の半導体等製造装置に現れるホイストスケジューリング問題を扱う．ホイストスケジューリング問題とは，主に化学的処理を行うためのタンクを複数有する自動加工システムにおいて，ホイストと呼ばれる搬送機のスケジュールを求める問題であり，フローショップスケジューリング問題やジョブショップスケジューリング問題の拡張と捉えることができる．ホイストスケジューリング問題において，ジョブに対応する部品（製品）はあらかじめ定められた順序でタンクを移動する．部品ごとに，各タンクでの処理時間の上下限が与えられており，その範囲内であれば自由に処理時間を決定できるが，処理終了後は，直ちに部品をタンクから取り出し，次のタンクに搬送・収納しなくてはならない．また，各タンクにバッファはなく，同時に収納できる部品は高々 1 つであるため，タンクに部品が収納されている場合，それを取り出さない限り次の部品を収納することはできない．タンクからの部品の取出し，次のタンクへの搬送・収納はホイストで行われ，それぞれのホイストは複数の部品を同時に処理することはできない．

ホイストスケジューリング問題は，その設定や条件によっていくつかのタイプに分類され，それぞれについてさまざまな手法が提案されている [3]．たとえば，単一アイテム（製品の種類が 1 つ）の問題に対して，ある一定のスケジュールを繰り返す，サイクリックスケジュールの中で最適なものを求めるための厳密解法や，複数種類の製品を扱う問題に対する発見的手法などが提案されている（文献 [4, 5] など）．

本稿で扱う問題は特殊な条件を有していることに加え，数千の作業のスケジュールを数分程度で出力することが求められている．そのため，汎用ソルバーの利用や厳密解法の適用は実用上困難であると判断し，問題に特化した発見的手法の提案を行う．

2. 問題設定

本節では，本稿で扱う問題について述べる．ただし，議論を簡潔にするため説明を簡略化しており，考慮すべき設定や条件が実際には種々存在する．

ジョブの集合を J，タンクの集合を M，ホイストの集合を H とする．各ジョブ $j \in J$ はあらかじめ定められた順（投入順）で投入口に投入され，k_j 個のタンク $m_{j1}, \ldots, m_{jk_j} \in M$ で順に処理された後，搬出口に搬送されて回収される．ジョブ j のタンク m_{jk} での処理時間 p_{jk} には，下限 p^-_{jk} と上限 p^+_{jk} が定められており，$p^-_{jk} \leq p_{jk} \leq p^+_{jk}$ でなくてはならない．ここで，$m_{j0}, m_{j(k_j+1)}$ をそれぞれ投入口，搬出口として，「タンク m_{jk} からジョブ j を取り出し，タンク $m_{j(k+1)}$ に搬送・収納する一連の作業」を 1 つの作業とみなして O_{jk} で表す（図 1）．また，各作業 O_{jk} について，O_{jk} を行うホイスト $h_{jk} \in H$ はあらかじめ定められており，所要時間は所与のものとする（ホイストはジョブを取り出した後，直ちに次のタンクに移動するため，O_{jk} の所要時間はホイストの動作スケジュールによらない）．

タンクおよび投入口と搬出口は直線状に配置されて

ののべ　こうじ
法政大学デザイン工学部
〒 162–0843 東京都新宿区市谷田町 2–33

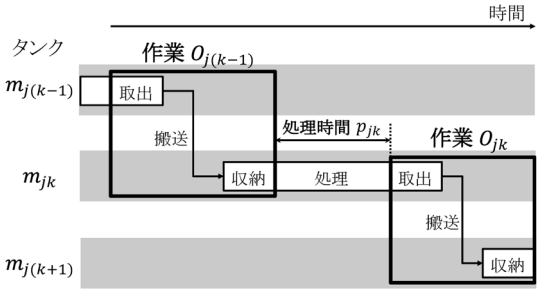

図 1　作業の定義

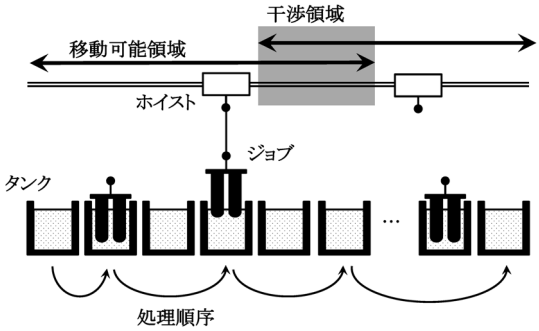

図 2　製造装置概略図

おり，各ホイストが処理可能なタンク（ジョブを取り出したり収納したりすることのできるタンク）は直線上の連続した区間にあるものとする．また，すべてのホイストは同一レール上を動き，互いに追い越すことはできず，隣り合うホイストは移動可能領域が重なるため，互いに衝突しないよう考慮する必要がある（以下，移動可能領域の重なる部分を干渉領域と呼ぶ）．製造装置の概略図を図 2 に示す．

本稿で扱うホイストスケジューリング問題は，以下の制約を満たすように，各ジョブ j の各作業 O_{jk} の処理開始時刻 s_{jk} を決定する問題と捉えることができる：

(a) 作業 $O_{j(k-1)}$ の完了時刻から作業 O_{jk} の開始時刻までの時間（ジョブ j のタンク m_{jk} での処理時間 p_{jk} に対応）が，所定の範囲 $[p^-_{jk}, p^+_{jk}]$ 内にある．

(b) ホイストが連続して処理する 2 つの作業 O_{jk}, $O_{j'k'}$ 間に，O_{jk} の収納タンクから $O_{j'k'}$ の取出タンクへ移動する時間が確保されている．

(c) 複数のジョブが同時に同じタンクに収納されていない．

(d) ホイストが干渉領域内のタンクに対して処理（取出し，または収納）を行う場合，干渉領域に入る移動を開始してから，処理を終えて干渉領域から出る移動を開始するまでの間，隣接するホイストはその干渉領域に入る移動を行わない．

また，本稿では，最大完了時刻の最小化を目的とする．

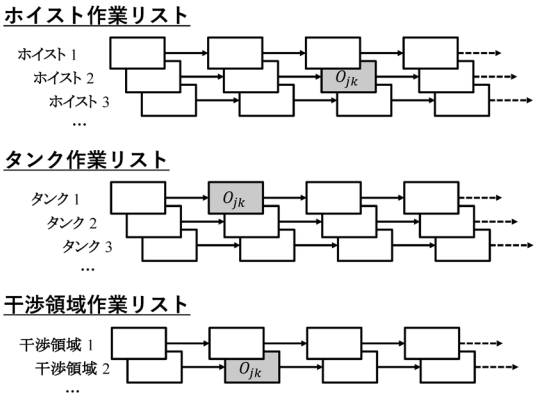

図 3　作業リスト

3. 提案アルゴリズム

前節の制約 (a) は，ある 2 つの作業 $O_{j_1k_1}, O_{j_2k_2}$ の開始時刻の差に関する不等式

$$s_{j_2k_2} - s_{j_1k_1} \geq d_{j_1k_1j_2k_2} \tag{1}$$

を用いて記述することができる．（処理時間の下限制約と上限制約のそれぞれを不等式で記述する．上限制約の場合，右辺定数の値は負となる．）その他の制約 (b)〜(d) についても，各ホイストについて当該ホイストが処理する作業の処理順序，各タンクについて収納先が当該タンクである作業の処理順序，各干渉領域について搬送領域が当該干渉領域と重なる作業の処理順序をそれぞれ定めれば，各制約を不等式 (1) の形で記述でき，定められた処理順序に違反しない範囲で実行可能なスケジュールが存在するか否かの判定と，存在する場合には，最大完了時刻を最小にするスケジュールを多項式時間で計算することができる．具体的には，各作業を節点とするネットワークを考え，不等式 (1) のそれぞれを節点 $O_{j_1k_1}$ から節点 $O_{j_2k_2}$ への長さ $d_{j_1k_1j_2k_2}$ の枝に対応させて，長さが正の閉路が存在するか否かを判定すればよい．存在する場合，定められた処理順序に従う実行可能スケジュールは存在しないことを意味し，正の閉路が存在しない場合は，最初のジョブの最初の作業から他の各作業への最長路が計算でき，その長さが最大完了時刻を最小にする開始時刻となる．以下では，ホイストごと，タンクごと，干渉領域ごとの作業の処理順序をリストで表し，それぞれをホイスト作業リスト，タンク作業リスト，干渉領域作業リストと呼び，すべてをまとめて作業リストと呼ぶ（図 3）．

ホイストスケジューリング問題は，各作業の開始時刻を決定する問題であるが，本稿で提案するアルゴリズムは，各作業の開始時刻を直接求めるのではなく，最適なスケジュールが得られる作業リストを求めようとするものであり，実行可能な作業リストを生成する構築フェーズと，最大完了時刻が早くなるよう作業リストを修正する改善フェーズから成る．

3.1 構築フェーズ

構築フェーズでは，上述のネットワークを保持・更新しながら，長さが正の閉路が存在しないように，所与の投入順に従ってジョブごとに作業を一つずつ各作業リストに挿入していく．ネットワークは初めすべての節点と制約 (a) を表す枝のみで構成され，作業を作業リストに挿入することは，制約 (b)〜(d) を表す枝をネットワークに追加することに対応する．

作業リストへの挿入は，タンク作業リスト，ホイスト作業リストの順に行い，搬送領域が干渉領域と重なる場合には，最後に干渉領域作業リストへの挿入を行う．各作業リストへの挿入位置は，ジョブの投入順，およびジョブごとに定められた作業の処理順序に違反しない範囲で最早の位置から始め，枝の追加によって正の閉路が生成される場合（実行不可能となる場合）には，挿入位置を一つ後ろにずらす．正の閉路が生成されない場合は，最長路の計算によって開始時刻を更新する．タンク作業リストとホイスト作業リストの両方（搬送領域が干渉領域と重なる場合は，干渉領域作業リストを含めた 3 つの作業リスト）への挿入が完了した時点で，次の作業に移る．一方，実行可能な挿入位置が存在しない作業リストが一つでもある場合には，一つ前の作業にバックトラックし，挿入位置を後ろにずらす．

なお，ジョブごとに作業を一つずつ追加していき，実行不可能となる場合にはバックトラックを行うというアイデアは，文献 [5] で提案されている手法においても用いられている．

3.2 改善フェーズ

改善フェーズでは，クリティカルパスの情報を利用しながら局所探索によって作業リストの改善を試みる．ただし，実行可能な作業リストを効率的に探索するため，近傍探索において作業リストを直接修正することはせずに，クリティカルパス上の各枝 $(O_{jk}, O_{j'k'})$ $(j \neq j')$ に対して，以下のように構築フェーズを再度実行することで作業リストを生成する．ジョブ j のほうがジョブ j' よりも投入順が後の場合には，作業 O_{jk} を作業リストに挿入する際，$O_{j'k'}$ の直前に挿入することを禁止して（$O_{j'k'}$ を O_{jk} の挿入禁止位置として設定して）構築フェーズを実行する．また，j' のほうが j よりも投入順が後で，$O_{j'k'}$ に挿入禁止位置が設定されている場合には，これを解除して構築フェーズを実行する．これらの操作によって，$O_{j'k'}$ の開始時刻が早まる可能性があり，その結果，最大完了時刻が早まることを期待している．

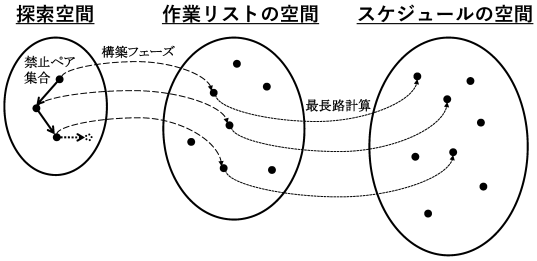

図 4　改善フェーズにおける局所探索

改善フェーズにおける局所探索の概念図を図 4 に示す．作業 O_{jk} とその挿入禁止位置 $O_{j'k'}$ の組（以下，禁止ペアと呼ぶ）の集合が局所探索における解であり，それらの集合（集合族）が探索空間となる．空集合を初期解とし，クリティカルパスの情報に基づいて禁止ペアを追加したり削除したりすることで解を改善していく．解の評価のたびに構築フェーズによって作業リストが生成され，最長路計算で得られるスケジュールの最大完了時刻が解の評価値となる．

4. アルゴリズムの実装と高速化

4.1 実行可能性の判定と開始時刻の更新

構築フェーズにおいて，作業リストに作業を挿入（ネットワークに枝を追加）しようとするたびに実行可能性（正の閉路が生成されるか否か）の判定を行い，挿入後には開始時刻（最長路長）の値を更新する必要がある．これらの計算時間を削減するため，開始時刻の計算と更新には FIFO ラベル修正法 [6] を，実行可能性の判定にはダイクストラ法を用いる．

例として，作業 O_{jk} を作業リストの O' と O'' との間に挿入することを考える．このとき，ネットワークに 2 本の枝 $(O', O_{jk}), (O_{jk}, O'')$ を追加することになる（図 5）．これらの枝を追加することで生じ得る閉路は，(i) O' から (O', O_{jk}) を経由し，O_{jk} から O' に至る路によって O' に戻る閉路，(ii) O_{jk} から (O_{jk}, O'') を経由し，O'' から O_{jk} に至る路によって O_{jk} に戻る閉路，(iii) O' から (O', O_{jk}) と (O_{jk}, O'') を経由し，O'' から O' に至る路によって O' に戻る閉路のいずれかに分類される．(i) については，O_{jk} から O' への最

長路長（O_{jk} から O' に至る路が存在しない場合は $-\infty$ とみなす）と (O', O_{jk}) の長さ d' の和が 0 以下であれば，正の閉路は生成されないことになる．(ii),(iii) についても同様に，O'' から O_{jk} に至る最長路長，O'' から O' に至る最長路長がわかれば正の閉路が生成されるかどうかの判定ができる．この最長路長の計算をダイクストラ法で行う（最長路長を求めるため，枝の長さの正負を反転させ，さらに，すべての枝の長さが正になるよう変換しておく）．

枝の追加後は，追加した 2 本の枝の始点 O',O_{jk} をキューに格納した状態でラベル修正法を適用することで，開始時刻を更新できる．このとき，更新前の開始時刻を保持しておくことで，追加した枝をバックトラックのときに削除する際，開始時刻の情報を復元することができる．

4.2 逐次最適化

計算時間を削減するため，すべてのジョブを対象として一度にスケジュールの構築・改善を行うのではなく，あらかじめ指定した数（K とする）の連続するジョブをひとまとまりとして最適化を逐次行いながら，作業の処理順序をジョブ単位で固定していく．まず，先頭の K 個のジョブについて，構築フェーズと改善フェーズを実行し，作業リストを得る．次に，$(K+1)$ 番目のジョブを挿入した後，2〜$(K+1)$ 番目の K 個のジョブを変更対象として，作業リストの改善を行う．以下同様に，作業リストにジョブを一つずつ挿入しながら，最後の K 個を最適化の対象として（最後の K 個以外のジョブについては処理順序を固定して）改善フェーズを実行することを繰り返す．

この逐次最適化の方法を用いると，最初に投入するジョブ数個分の計算が完了した時点で装置へのジョブの投入を開始し，装置を稼働させながら残りのジョブの計算を行うことが可能となる．これにより，最適化計算開始後，装置が稼働するまでの時間を短縮させることができる．また，本稿では装置に投入するジョブの集合 J を所与のものとしているが，実運用においては，ジョブの処理中に新たなジョブが追加される状況も考慮する必要がある．逐次最適化によって，このような動的な状況にも対応可能となる．

4.3 開始時刻の固定

前節の逐次最適化において，ある程度計算が進んだ状況を考えると，計算初期に挿入されたジョブについては，新たなジョブの挿入による影響は少なく，作業の開始時刻はほとんど変化しないと考えられる．そこで，アルゴリズムの計算過程において，一定期間（一定数のジョブが挿入される間），開始時刻が更新されなかったジョブについては開始時刻を固定することで，最長路計算にかかる時間の削減を図る．

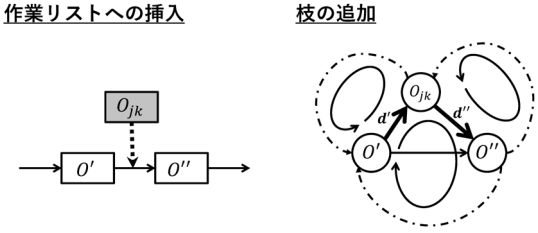

図 5　作業リストへの作業の挿入

5. 計算実験

提案アルゴリズムを C++ で実装し，実データに基づいて作成された 5 つの問題例 1〜5 を用いて計算実験を行った．問題例の規模はすべて共通であり，ジョブ数は 75，各ジョブの作業数は 14〜18，ホイスト数は 2 である．

なお，計算実験に用いた計算機の CPU は Core i7-

表 1　一括最適化と逐次最適化との比較

問題例	一括最適化	逐次最適化				
		$K=1$	$K=3$	$K=5$	$K=10$	$K=15$
1	427200	427200	427200	427200	427200	427200
	51.4(s)	1.5(s)	1.9(s)	2.4(s)	3.3(s)	19.0(s)
2	59579	59579	59579	59579	59579	59579
	327.3(s)	3.5(s)	10.8(s)	22.4(s)	72.4(s)	238.3(s)
3	175219	175142	175142	175154	175174	175174
	103.0(s)	2.8(s)	4.9(s)	8.3(s)	24.4(s)	64.0(s)
4	143274	143274	143274	143274	143274	143274
	397.7(s)	3.6(s)	12.5(s)	23.8(s)	77.8(s)	263.9(s)
5	72333	58293	55899	56086	56086	56086
	5993.3(s)	4.7(s)	9.7(s)	18.2(s)	80.5(s)	291.2(s)

表 2　開始時刻固定による効果

問題例	開始時刻の固定		B/A
	なし (A)	あり (B)	
1	427200	427200	1.00
	4.2(s)	1.9(s)	0.45
2	59579	59579	1.00
	41.7(s)	10.8(s)	0.26
3	175178	175142	1.00
	10.4(s)	4.9(s)	0.47
4	143274	143274	1.00
	90.1(s)	12.5(s)	0.14
5	55899	55899	1.00
	70.8(s)	9.7(s)	0.14

6700K 4.00 GHz, RAM は 32 GB であり, OS は Windows 10 Pro である.

5.1　逐次最適化の効果

まず，4.2 節で述べた逐次最適化の効果を確認するため，75 ジョブのすべてを対象に構築フェーズと改善フェーズを実行した場合（一括最適化）と，$K = 1, 3, 5, 10, 15$ として逐次最適化を行った場合との比較を行った．計算結果を表 1 に示す．各問題例のそれぞれの場合について，計算で得られたスケジュールの最大完了時刻と計算に要した時間（秒）がそれぞれ上段と下段に記されている．

問題例 1〜4 については，一括最適化か逐次最適化かの違いや K の値の違いが最大完了時刻に与える影響はまったくないか，非常に小さいことがわかる．これは，構築フェーズのみで十分良質のスケジュールが得られているためと考えられる．実際，一括最適化において，構築フェーズで得られたスケジュールが改善フェーズで更新された回数（局所探索における解の移動回数）は，問題例 1〜4 についてはそれぞれ 0 回，5 回，12 回，0 回であった．最終的に出力されるスケジュール自体は，計算開始後それぞれ 1.7 秒，5.7 秒，27.4 秒，7.4 秒で得られており，残りの計算時間は，局所探索において，現在のスケジュールよりも良いスケジュールが近傍内に存在しないことを確認することに費やされていたことになる．一方，問題例 5 については，逐次最適化を行うことで，一括最適化の場合よりも良質のスケジュールが得られている．問題例 5 は，他の問題例と比べてタンクでの処理時間が短く，複数の作業がホイストを取り合う競合が発生しやすい傾向にある．一括最適化では，構築フェーズで得られるスケジュールに最適化の余地が残されているものの，最適化の対象となるジョブ数が多いために，提案手法の局所探索では十分最適化しきれていないといえる（改善フェーズにおけるスケジュールの更新回数は 66 回であった）.

計算時間に関しては，逐次最適化を行うことによって削減できること，および K の値を小さくするほど削減効果は大きくなることが，すべての問題例において確認できる．K をどの値に設定すべきかは問題例によって異なるため事前に判断することは困難であるが，計算時間を考慮すると，K を比較的小さな値に設定して逐次最適化を行うことが，現実的には有用と考えられる.

5.2　開始時刻固定の効果

次に，4.3 節で述べた開始時刻の固定による計算時間削減の効果を確認するための比較実験を行った．各問題例に対して，開始時刻の固定を行わない場合と行う場合の両方について，$K = 3$ として逐次最適化を実行したときの計算結果を表 2 に示す．それぞれの問題例について，最大完了時刻を上段，計算時間（秒）を下段に記し，開始時刻の固定を行う場合の，行わない場合に対する比を最後の列に記載している.

すべての問題例に対して，最大完了時刻を悪化させることなく，計算時間が大幅に削減されていることが確認できる．なお，問題例 3 については，開始時刻を固定することで最大完了時刻も改善されているが，これは，改善フェーズにおける局所探索の探索経路が変化したためで，開始時刻固定による直接的な効果ではないと推察される.

5.3　現行スケジューラとの比較

最後に，従来用いられているスケジューラとの比較について述べておく．現行スケジューラは，シミュレーションベースの手法であり，パラメータの値を変化させながらシミュレーションを指定時間繰り返し実行する中で得られる最良のスケジュールを採用するというものである．提案手法は，最大完了時刻最小化の意味で，現行スケジューラとほぼ同程度以上の質のスケジュールを得ることができている.

なお，現行スケジューラには，シミュレーションに必要なパラメータが多数存在する．そのため，装置に入力される問題例の傾向をある程度想定したうえで，最適パラメータの探索を事前に数時間かけて行い，実行時に試すパラメータの値の範囲を絞り込んでいる．提案手法はこの手間が不要であり，現行スケジューラに対して優位な点であるといえる.

6.　おわりに

本稿では，現実の半導体等製造装置を対象としたホ

イストスケジューリング問題に対して，ネットワークアルゴリズムと局所探索法を用いた発見的手法の提案を行った．また，提案手法を高速化するための工夫を加え，その効果を確認した．

今後，実証用装置を用いた検証を行う予定である．

参考文献

[1] 野々部宏司，“ホイストスケジューリング問題に対する発見的手法，”日本オペレーションズ・リサーチ学会 2018 年春季研究発表会アブストラクト集，pp. 186–187, 2018.

[2] 野々部宏司，“ホイストスケジューリング問題に対する発見的手法，”スケジューリング・シンポジウム 2018 講演論文集，pp. 139–143, 2018.

[3] C. Bloch, M.-A. Manier, P. Baptiste and C. Varnier, “Hoist scheduling problem,” *Production Scheduling*, P. Lopez and F. Roubellat (eds.), John Wiley & Sons, Chapter 8, 2013.

[4] A. El Amraoui and M. Elhafsi, “An efficient new heuristic for the hoist scheduling problem,” *Computers & Operations Research*, **67**, pp. 184–192, 2016.

[5] H.J. Paul, C. Bierwirth and H. Kopfer, “A heuristic scheduling procedure for multi-item hoist production lines,” *International Journal of Production Economics*, **105**, pp. 54–69, 2007.

[6] R.K. Ahuja, J.B. Orlin and T.L. Magnanti, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.