マルチコアマシンにおける
並列 A*探索の探索オーバーヘッドの解析と
アルゴリズムの再評価

陣内　佑
東京大学教養学部学際科学科

指導教員　福永 Alex

2015 年 1 月 22 日

概要

グラフ探索問題のうち最も重要なものの一つとして、辺にコストのついた有向グラフ、初期状態、ゴール状態を入力とし、初期状態からゴール状態までの経路のうちコストが最小である経路を出力する問題がある。この問題を解く為の代表的なアルゴリズムの枠組みとして A*探索がある。この A*探索を並列化する手法は数多く提案されている。

HDA*は特に分散メモリ環境において効率的な並列 A*探索である。HDA*は各スレッドに対してオープンセット・クローズドセットをローカルに置き、スレッド間で非同期的に仕事を分配する。HDA*は展開済のノードの全てをクローズドセットに保存する為、TDS などの展開済ノードの一部しか保存しないアプローチと比較して余分な展開ノードが少ない。しかし HDA*は PBNF と比較して余分な展開ノードが生じると指摘されている。PBNF は展開ノード数を小さくすることを主眼に設計された手法であり、共有メモリ環境において HDA*よりもパフォーマンスに優れるとされている。しかしながら、先行研究による HDA*と PBNF の性能比較実験は複数問題点がある。

本研究はまず、HDA*の余分な展開ノードを定性的に分析し、その原因を 3 つに分類する。また、問題サイズが十分に大きい場合、それらのオーバーヘッドが無視出来る大きさであることを分析的・実験的に示す。次に、先行研究における HDA*と PBNF の性能比較実験の問題点を指摘し、その上でより詳細な性能比較実験を行う。その実験結果として、 (1) 両アルゴリズムで展開ノード数は大きく変わらないこと、 (2) 逐次実行で 10 秒程度以上の問題では HDA*の方がパフォーマンスに優れることを示す。

目次

1 序論

1.1 研究の背景

グラフは情報科学に通底する問題ドメインであり、グラフを解き明かす探索アルゴリズムは分野の基本骨子をなしている。人工知能の分野においても探索は基盤となるアルゴリズムの一つであり、高速で効率的な探索アルゴリズムを研究することは人工知能ひいては情報科学全体に意義のある研究である。近年単一プロセッサの高速化は電力の限界を迎え、並列化による高速化へとパラダイムが移った。ハードウェアは今後も各 CPU の計算速度の発展よりも並列化による高速化が進むだろうと予測されている。故に今後ハードウェアの高速化を活用する為にはこの並列システムを利用したアルゴリズムを開発する必要がある。探索アルゴリズムも単一プロセッサ上だけではなく、マルチコア・分散システムにおいてもそれぞれ効率的なアルゴリズムを研究しなければならない。

1.2 関連研究

グラフを探索するアルゴリズムのうち最も重要なものの一つとして A*(エースター) アルゴリズムがある [1]。この A*アルゴリズムを並列化する手法は数多く提案されている。

Kumar らはオープンセット、クローズドセットをスレッド間で共有して並列に探索を行う Parallel A*を提案した。Parallel A*は非常にシンプルな手法であるが、データ構造を共有するために同期オーバーヘッドが非常に大きいという問題がある。5 章で見るように、このシンプルな手法は逐次 A*よりも遅くなってしまう。

オープンセット、クローズドリストは同期の必要がない lock-free なデータ構造を用いて実装することが出来る。lock-free なデータ構造とは、複数のスレッドが同時にアクセスすることの出来るデータ構造である。このデータ構造はスレッドの同期の必要がないが、アクセス速度が遅い。Burns らはオープンセット、クローズドリストを lock-free なデータ構造で実装をした lock-free parallel A*を実装し、この手法も逐次 A*よりも遅くなってしまうことを示した。

同期オーバーヘッドを解決する手法として、Kishimoto らは Hash Distributed A* (HDA*) を提案した [2]。HDA*では、各スレッドがそれぞれローカルにオープンセットとクローズドセットを持つ。グラフのノードはハッシュ関数によって均等かつ無作為に分割し各スレッドに割り当てられる。各スレッドは自分に割り当てられたノードのみを展開し、他スレッドに割り当てられたノードを生成した場合はそのスレッドにノードを非同期的に送信する。HDA*は各スレッドがローカルのデータ構造にしかアクセスしない為、同期オーバーヘッドが殆どない。その為、HDA*は特に分散メモリ環境で効率的であるとされている。

Parallel Best-NBlock-First (PBNF, SafePBNF) は余分な展開ノード数を小さくすることを目的として設計された共有メモリ環境における並列探索アルゴリズムである [3]。理想的には、全てのスレッドは最も有望なノードを探索する。PBNF はこれを実現するために、状態空間を複数の nblock に分割し、それぞれのスレッドが最も有望なノードを持つ nblock を占有して探索する。PBNF は分散メモリ環境では有効ではないが、現在共有メモリ環境で最もパフォーマンスの良い state-of-the-art の手法であるとされている。

Burns らは共有メモリ環境で PBNF と HDA*を含めた 9 つの並列アルゴリズムの性能比較を行い、その中で PBNF と HDA*を最も有望な (高速な) アルゴリズムとした。その上で PBNF は HDA*よりパフォーマンスに優れると結論をした。その理由として、PBNF が HDA*よりも展開ノード数が小さいという結果があったことを指摘した。しかしながら、Burns らの比較実験には二つ問題点がある。一つに、Burns らの実験にお

ける HDA*のハッシュ関数の実装は最適なものではなかった。もう一つは、ベンチマーク問題のサイズが小さいということが挙げられる。並列アルゴリズムは初期化に時間がかかる為、ある程度問題サイズが大きいものでないと実質的な高速化効率を正確に測ることが出来ない。よって、適切な実験設定の元で両アルゴリズムを再評価することは有意義である。一方、HDA*の展開ノード数の増加の原因はまだ研究されていない。HDA*だけでなく、並列探索アルゴリズムの展開ノード数の増加の原因を定性的に分析する事例は非常に少ない。

1.3 研究の目的

本研究の目的は 2 つある。

1. HDA*の展開ノード数の増加の原因を分析する。

   Burns らは PBNF と比較して HDA*は余分な展開ノードが生じると主張した。本研究はまず、HDA*の展開ノード数の増加を定性的に分析し、その原因を分類する。ここで得られた知見を元に、実験的に展開ノード数の定量的な分析を行う。

2. 共有メモリ環境において、HDA*と SafePBNF の性能評価を行う。

   Burns らの性能評価実験には二点問題があった。それらの問題点を改善した上でより詳細な比較実験を行う。まず、HDA*と SafePBNF の実装の詳細がパフォーマンスに与える影響を調べる。それらの結果を踏まえ、最適な HDA*と SafePBNF の実装をもって性能の比較を行う。

1.4 本論文の構成

本論文は以下の通りに構成される。2 章で A*探索の詳細と A*探索の並列化手法について述べる。3 章は本研究で扱うベンチマーク問題の問題ドメインについて解説する。4 章は HDA*の探索ノード数の増加を解析する。5 章では HDA*と SafePBNF のチューンアップを行ったのち、性能の比較評価を行う。6 章は本研究で行った実験・解析から得られる知見をまとめる。

2　A*探索とその並列化

2.1　A*探索

A*探索アルゴリズム (A*, A*探索) はヒューリスティック関数によって、展開されたノードの中から最も有望であるノードを推定し、そのノードを優先して探索するアルゴリズムである [1]。ヒューリスティック関数を利用しないブラインド探索と比較して高速に最適解を見つけられることが多い。A*探索は以下のようなアルゴリズムである。

1. 初期状態をオープンセットに加える。
2. オープンセットにあるノードの中でゴール状態までの見積りが最小であるノードを展開する。
   2.1. 展開したノードをオープンセットから取り除き、クローズドセットにその状態を加える。
   2.2. 展開した状態がゴール状態かどうかを調べる。ゴール状態である場合は解を出力して終了する。
   2.3. 目標状態でない場合、展開した状態から 1 つの行為で遷移可能な状態のうち、クローズドセットにないすべての状態をオープンセットに加える。
3. オープンセットが空になった場合は解なしとして終了する。

オープンセットは未展開のノード、**クローズドセット**は展開済のノードが保存されるデータ構造である。あるノード n からゴール状態までの見積り $f(n)$ は $f(n) = g(n) + h(n)$ で計算される。$g(n)$ は初期状態からノード n までの経路にかかったコストである。$h(n)$ は状態からゴール状態までの距離を推定する**ヒューリスティック関数**である。ヒューリスティック関数が **admissible** であるならば、A*探索の解の最適性が保証される。ヒューリスティック関数が admissible であるとは、ゴール状態までの推定コストを過小に見積もることがないということである。

オープンセットのデータ構造はドメインに応じて様々なものが選択される。heap はあらゆるドメインに対して汎用的に使うことの出来る実装である。heap へのアクセスにかかる時間は $O(\log n)$ である。一方、ドメインのコストが全て整数であれば nested bucket で実装することが出来る。nested bucket は f と h の取りうる組み合わせの全てに対して一つの配列を用意する。配列に対しては $O(1)$ のコストでアクセスが可能であるので、nested bucket は $O(1)$ でアクセスが出来る。すなわち、nested bucket の方が heap よりも高速であるが、適応出来るドメインが狭い。本研究は nested bucket と heap の両方で実験を行う。以降の議論では、ここで述べた A*探索を、他の並列探索手法と区別する為に**逐次 A***と呼ぶ。

2.2　A*探索の並列化にかかるオーバーヘッド

並列探索は、理想的にはスレッドの数だけ高速化する。しかしながら、一般に並列探索には、並列化に伴うオーバーヘッドが生じる。並列化のオーバーヘッドは以下の 3 つに分類される。

1. 探索オーバーヘッド

   一般に並列探索は逐次 A*よりも多くのノードの展開を必要とする。並列化に伴い余分に展開したノードの割合を**探索オーバーヘッド**と呼ぶ。本研究では、探索オーバーヘッドは以下の式によって推定する。

$$SO := \frac{\text{並列実行によって展開したノード数}}{\text{逐次実行によって展開したノード数}} \tag{1}$$

SO は理想的には 1 になるが、多くの場合 1 よりも大きくなる。また、並列化によって展開ノード数が小さくなる場合もある。このとき、SO は 1 よりも小さくなる。

2. 同期オーバーヘッド

   同期オーバーヘッドは、他スレッドの処理を待つ為にアイドル状態で待たなければならない場合に生じる。データ構造の一部は同時に複数のスレッドがアクセスすることが出来ない。そのようなデータ構造をスレッド間で共有する場合は**相互排他ロック (ロック)** を必要とする。相互排他ロックは、ロックを獲得したスレッド以外のスレッドによるアクセスを禁止する制御機構である。あるスレッドが他スレッドによって獲得された相互排他ロックが解放されるまで待つ場合、同期オーバーヘッドとなる。相互排他ロックを必要としないか、ロックの解放を待たずに他の処理に移る場合を**非同期通信**と呼ぶ。

3. 通信オーバーヘッド

   通信オーバーヘッドはスレッド間の通信にかかる遅延である。スレッド間の通信は仕事の分配や情報の共有の為に行われる。

これらのオーバーヘッドはトレードオフの関係にある。理論的にこれらを最小にすることは非常に困難である為、多くの場合は実験的に最適な実装にチューニングが行われる。

並列化にかかるオーバーヘッドの計測の為には**高速化効率 (efficiency)** という指標が用いられる。Efficiency は以下のように定義される。

$$\text{Efficiency} := \frac{\text{逐次実行による実行時間}}{\text{並列実行による実行時間} \times \text{スレッドの数}} \tag{2}$$

Efficiency は理想的には 1 になるが、上述のオーバーヘッドがかかる為に多くの場合 1 未満である。Efficiency が 1 の場合を **Perfect linear speedup** と呼ぶ。Efficiency が 1 を越える場合を **Superlinear speedup** と呼ぶ。

本研究では探索オーバーヘッドと高速化効率を主な指標として用いる。

2.3 Parallel A*

最もシンプルな A*の並列化手法は Kumar らの提案した **Parallel A***である [4]。Parallel A*は各スレッドで A*探索を行う。スレッド間でオープンセット、クローズドセットを共有することで並列に探索を行う。この手法はオープンセット、クローズドセットのデータ構造を保つ為に相互排他ロックを必要とする。各スレッドはノードを展開する度にオープンセットに、ノードを生成する度にクローズドセットにロックを獲得してアクセスする必要がある。その為 Parallel A*は同期オーバーヘッドが非常に大きいという問題がある。5 章で見るように、このシンプルな手法は逐次 A*よりも遅くなってしまう。以降、Parallel A*は並列 A*探索一般ではなく、特にこの手法を指す。

2.4 Parallel Retracting A* (PRA*)

Parallel Retracting A* (PRA*) は各スレッドにローカルにオープンセットとクローズドセットを持つ [5]。グラフのノードはハッシュ関数によって均等かつ無作為に分割し各スレッドに割り当てられる。ハッシュ関数はノードの状態に対して担当のスレッドを示すハッシュ値を返す。各スレッドは自分に割り当てられたノードのみを探索し、他スレッドに割り当てられたノードを生成した場合はそのスレッドにノードを同期的に送信する。ハッシュ関数が均等かつ無作為な分割であれば、仕事の分配は均等であり、全てのスレッドが効率的に仕事をすることが出来る。また、クローズドセットをローカルに置くことが出来るので、分散メモリ環境においても効率的に重複の確認が可能である。ノードの送信時、送信側のスレッドは受信側のスレッドが受信の態勢に入るのを待ち、受信側スレッドは受信に成功したら送信側スレッドにそれを報告する。送信側スレッドはその報告を受け取るまで待機しなければならない。この為 PRA*は同期オーバーヘッドが大きいという問題がある。

Mahapatra と Dutt は PRA*で提案されたハッシュ関数による仕事の分配をより深く研究した [6]。彼らはまず、PRA*で提案されたハッシュ関数による分配を SEL_SEQ_A*に適応したアルゴリズムを **Global Hashing of Nodes (GOHA)** と定義して実装をした。SEL_SEQ_A*は A*探索の派生であり、ノードの展開時に f 値の大きいノードの生成を後周しにする手法である。その上で、彼らは GOHA の発展として 2 つの手法を提案した。一つは GOHA に Quality Equalizing のメカニズムを適応した **Global Hashing of Nodes and Quality Equalizing (GOHA&QE)** である。この手法は異なるメカニズムで重複の確認と仕事の分配を行う。一度オーナーに重複がないことが確認されたノードは、どのプロセスで展開しても問題ない。GOHA&QE は仕事を効率的に分配する為に、重複の確認後に更に各プロセスの仕事の質に応じて仕事の再分配を行う。GOHA 及び GOHA&QE で使われるハッシュ関数はグローバルであり、ノードはあるプロセッサからどのプロセッサにも送信される可能性がある。一般に、並列コンピュータアーキテクチャにおいて、プロセッサ同士の通信コストはペア毎に異なる。よって、このようなハッシュ関数は最適ではない可能性がある。**Local Hashing of Nodes and Quality Equalizinig (LOHA&QE)** は探索空間を複数のグループに分割し、隣り合うグループが隣り合うプロセッサに割り当てる手法である。Mahapatra と Dutt は Travelling Salesperson Problem においてこれらの手法が単純な PRA*よりもパフォーマンスに優れることを示した。この手法はコンピュータのアーキテクチャと問題ドメインに対して複数の前提を必要とするという問題点がある。

2.5 Transposition-table driven work scheduling (TDS)

Transposition-table driven work scheduling (TDS) は分散メモリ環境における IDA*の並列アルゴリズムである [7]。**Iterative Deepening A* (IDA*)** はメモリ効率に優れた A*探索の派生である [8]。IDA*は反復的に深さ優先探索を行う。反復の度に探索したノードの情報を捨て、最大深度を大きくしながら、解が見つかるまで深さ優先探索を繰り返す。IDA*は A*探索と比較して必要なメモリの量が小さいが、反復の度に多くのノードを再展開する必要がある。TDS は PRA*と同様、状態に対するハッシュ関数を使ってノードを分配する。生成されたノードはハッシュ関数によって送信先のプロセスが決定される。ノードは Transposition-table で重複していないことを確認された場合に仕事のキューに加えられる。Transposition-

table は各プロセスがローカルに保持する。Transposition-table による重複の確認により、TDS の展開ノード数は IDA*よりも大きくならない。IDA*および TDS は繰り返し探索を行う為、反復の度に多くのノードを再展開する必要がある。PRA*などの A*探索をベースとする並列探索の場合はこのような反復的な再展開はない。TDS は、特に IDA*で再展開を必要とするノードの割合が非常に小さいことで知られる 15 Puzzle や Rubik's cube などの問題で効率的である。TDS のアイディアは主に二人零和ゲームに応用されている。

2.6 Hash Distributed A* (HDA*)

Hash Distributed A* (HDA*) は PRA*で提案された Hash による仕事の分配メカニズムに基づく手法である。HDA*は各スレッドにローカルにオープンセットとクローズドセットを持つ。グラフのノードはハッシュ関数によって均等かつ無作為に分割し各スレッドに割り当てられる。各スレッドは自分に割り当てられたノードのみを展開し、他スレッドに割り当てられたノードを生成した場合はそのスレッドにノードを非同期的に送信する。HDA*は同期オーバーヘッドが殆どなく、スレッド間で大きな情報の通信を必要としないので、特に分散メモリ環境で有効な手法である。

PRA*がノードの授受を確認する為に同期的な通信を行うのに対して、HDA*は非同期的にノードを送信する。その為 HDA*は PRA*と異なり、同期オーバーヘッドが殆どない。TDS と比較すると、TDS は探索深度を更新するイテレーション毎に多くのノードが再展開されるのに対して、HDA*にはこのような再展開はない。すなわち、HDA*は TDS と比較して展開ノード数が小さいというメリットがある。

HDA*は非常にシンプルであるという利点もある。HDA*は逐次 A*と比較してハッシュ関数による分配が必要なだけであり、各スレッドの処理は逐次 A*と概ね同じである。並列アルゴリズムはデバッグが難しい為、シンプルであるということは非常に価値がある。

HDA*の処理は以下の通りである。HDA*はまず、ルートプロセスで初期状態を展開する。その後、それぞれのスレッド P は最適解を発見するまで以下のループを繰り返す。

1. P は自分のメッセージキューに新しいノードが入っているかを確認する。もし入っていたら、それぞれのノードを P のクローズドセットと照合し、オープンセットに入れるべきかを確認する。
2. メッセージキューが空なら、P のオープンセットから最も優先度の高いノードを展開し、新しいノードを生成する。生成されたノード s に対してそれぞれハッシュ値 $K(s)$ を計算する。ノード s は $K(s)$ を担当するプロセッサのメッセージキューに送信される。この送信は非同期的で、ロックを必要としない。P は送信先からの返信を待たずに次の計算に移る。

ハッシュ関数は HDA*の性能に大きな影響を与える。ハッシュ関数に求められる要件は以下である。

1. 全てのスレッドに対してなるべく均等される。
2. 状態空間に対してなるべく**不偏に**均等な分配である。
3. ハッシュ値を高速に求めることが出来る。

以上の要件から、多くのドメインにおいて **Zobrist Hash** が使われる [9]。ノード s の状態が整数列 $\mathbf{x}$ で表現されるとする。状態の i 番目の整数の取りうる値は l_i 通りあるとする。ノードの状態を $\mathbf{x} = (x_0, x_1, \cdots, x_n)$ と置く。この時 x_i は $0 \leq x_i < l_i$ である。R_i を l_i 個の値を持つテーブルとする。R_i は予めランダムに

生成される。ここで Zobrist Function $Z(s)$ は以下のように定義される。

$$Z(s) := R_0[x_0]\ xor\ R_1[x_1]\ xor\ \cdots\ xor\ R_n[x_n] \tag{3}$$

R_i はランダムに生成される為、$Z(s)$ は均等な確率で値域の全ての値が出ると期待される。また、状態の全情報を利用して生成される為、局所的な状態の変化に対しても不偏な分配を行うことが出来る。xor によって計算する為、状態の親ノードからの差分のみを計算してハッシュ値を求めることが出来る。加えて、XOR 命令は非常に速い計算である。よって Zobrist Hash は高速に求めることが出来る。

2.7 Parallel Structured Duplicate Detection (PSDD)

Parallel Structured Duplicate Detection (PSDD) は PRA*とは異なる並列探索のフレームワークである [10]。PSDD はロックを取得する回数とスレッド間でノードを通信する回数を減らすことを目的として設計された。PSDD は structured duplicate detection (SDD) で提案されたアイディアに基づく [11]。SDD は abstract function によって複数のノードを 1 つのグループにまとめ、これを nblock と呼ぶ。それぞれの nblock はオープンセットとクローズドセットを持つ。nblock で展開されたノードの子が分配されうる nblock の集合を、その nblock の duplicate detection scope と呼ぶ。各スレッドは nblock の探索する時に、この duplicate detection scope を排他的に専有する。よって、PSDD はノードの展開時にロックを必要としない。ノードの展開は最も多い処理の一つであるので、これにロックを必要としないことは大きな利点である。加えて、生成されたノードはスレッドの専有する duplicate detection scope 内の nblock に配置される為、ノードの展開時に他のスレッドと通信をする必要がない。

2.8 Parallel Best-NBlock First (PBNF, SafePBNF)

PBNF search framework

```
while there is an nblock with open nodes do
  lock; b ← best free nblock; unlock
  while b is no worse than the best free nblock or we've done fewer than m expansions do
    n ← best open node in b
    if f(n) ≤ f(incumbent) then
      prune all open nodes in b
    else if n is a goal then
      incumbent ← n
    else
      for each child c of n
      insert c in the open list of the appropriate nblock
    end if
  end while
end while
```

Parallel Best-nblock First (PBNF) は PSDD のフレームワークを A*探索に適応したアルゴリズムで

ある [3]。PBNF は状態空間を複数の nblock に分割し、各スレッドが nblock を交換しながら探索する。それぞれの nblock はオープンセットとクローズドセットを持つ。それぞれのスレッドは最も有望なノードを持つ nblock を占有し、探索を行う。また、占有した nblock に隣接する nblock をスレッドの interference scope に加える。他のスレッドの interference scope に加わった nblock を探索することは出来ない。同一の nblock は複数の interference scope に入ることが出来ない。自分の占有している nblock よりも有望なノードを持つ nblock が空いていれば、今探索していた nblock と interference scope を解放し、新しい nblock を占有して探索を始める。このとき、あまりに頻繁な nblock の切り替えが生じるのを防ぐために、nblock を切り替える前に探索する最小のノード数 (最小展開ノード数) が設定される。これはチューニングの必要なパラメータである。nblock はスレッドに占有されるので、オープンセット・クローズドセットに相互排他ロックをかける必要がない。必要なロックは専有されていない nblock らを保存する heap のみである。PBNF は分散メモリ環境では有効ではないが、現在共有メモリ環境で最もパフォーマンスの良い state-of-the-art の手法であるとされている。PBNF の問題点として、その複雑さが挙げられる。すなわち、PBNF のパフォーマンスを引き出す為には空間を nblock に分割する方法・粒度と最小展開ノード数をドメインに対して最適化をする必要がある。

PBNF は **Livelock** が起こる可能性がある。Livelock は、複数のスレッドが同じ資源を解放する処理を実行しているにも拘らず、どちらのスレッドも資源が獲得出来ない状況である。この時、アルゴリズムは有限時間内に終了しないことになる。ある nblock が複数のスレッドの interference scope に入っている場合、どのスレッドもその nblock を探索することが出来ない状態になりうる。Burns らはこの問題を解決した **SafePBNF** を提案し、SafePBNF において Livelock が発生しないことを証明した。SafePBNF は Livelock を防ぐ為に nblock に対して **hot** というフラグを定義する。ある nblock がスレッドの interference scope に入っているとする。もしその nblock の最も有望なノードがそのスレッドの探索している nblock の最も有望なノードよりも有望である場合、この nblock を hot とする。hot な nblock を探索する要求があった場合、その nblock を interference scope に含むスレッドは現在探索している nblock を解放する。このメカニズムによって Livelock を防ぐことが出来る。

Livelock の可能性のあるアルゴリズムは完全ではない。アルゴリズムが完全であるとは、健全性と停止性を満たすことである。健全性とはアルゴリズムの結果が正しいことであり、停止性とはどのような入力に対しても有限時間内に結果を出力して停止することである。PBNF は Livelock が生じる可能性があるので、停止性を満たさず、よって完全ではない。SafePBNF は Livelock が起こらず、完全である。本研究で扱う A*、Parallel A*、HDA*は全て完全なアルゴリズムである。よって、同じ完全なアルゴリズムである SafePBNF を比較の対象とする。

3 問題ドメイン

3.1 Sliding-tile Puzzle (15 Puzzle, 24 Puzzle)

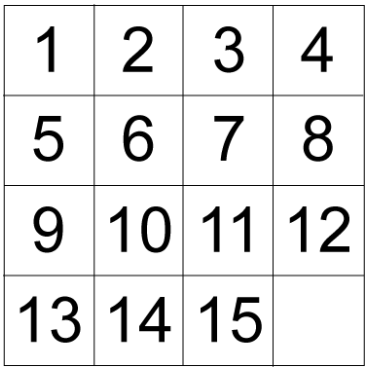


図 1: Sliding-tile Puzzle

図 1 のように、四角い枠にタイルが敷き詰められている。枠には一ヶ所タイルのない場所があり、そこに縦と横からタイルをスライドすることが出来る。**Sliding-tile Puzzle** は、タイルのスライドによって初期状態から特定のゴール状態まで動かす経路を求める問題である。15 Puzzle は 4×4 の盤に 15 枚のタイル、24 Puzzle は 5×5 の盤に 24 枚のタイルを置く。15 Puzzle の状態空間の大きさはおよそ 10^{13} 状態、24 Puzzle の状態空間はおよそ 10^{25} 状態である。 本研究では 15 Puzzle のヒューリスティック関数は**マンハッタン距離**を用いた。マンハッタン距離ヒューリスティックは、各タイルの現在位置からゴール状態での位置までのマンハッタン距離の総和で求められる。タイルは同時に 2 つ動かすことは出来ない。また、あるタイルがゴール位置に動かす為には少なくともマンハッタン距離の回数だけ動かす必要がある。よって、マンハッタン距離ヒューリスティックは admissible なヒューリスティック関数である。

マンハッタン距離を用いた 15 Puzzle はノードの展開が非常に高速なドメインである。ノードの展開が速いドメインは並列化によって高速化することが困難である。なぜならば、展開が高速である分、並列化オーバーヘッドによる遅延の影響が大きくなるからである。15 Puzzle ドメインにおいて高速化が可能であれば、多くのドメインにおいても同様に高速化が可能であると考えられる。この理由から、15 Puzzle は並列探索のベンチマーク問題として広く使われている。本研究でも主に 15 Puzzle をベンチマーク問題として使う。

マンハッタン距離は 24 Puzzle を解くにはシンプルすぎるヒューリスティックである。本研究では 24 Puzzle のヒューリスティック関数は **Disjoint pattern database heuristics** を用いた [12]。Disjoint pattern database heuristics は Culberson らの提案した Pattern databases を拡張したヒューリスティックである。Pattern database heuristics は問題の部分問題に対する最適解のコストを予めデータベースに記録し、ヒューリスティック関数として用いる手法である [13]。Disjoint pattern databases は複数の Pattern database を用いることで複数の部分問題のコストの和をヒューリスティック関数とする。Sliding-tile Puzzle においては、タイルを複数のグループに分ける。同じタイルは 2 つ以上のグループに含まれない。それぞれのグループにおいて、そのグループ内のタイルがゴール位置にたどり着く為の最小コストを求め、データベースに保存する。ある状態に対して、それぞれのグループがゴール位置にたどり着く為の最小コストをデータベースから読み、それらの和をヒューリスティックの値とする。この値は少なくともマンハッタン距離以上であ

る。マンハッタン距離がタイル間の相互作用を考慮しないのに対して、このヒューリスティックは同じグループ間の相互作用を考慮するので、多くの場合はマンハッタン距離よりも大きい。Disjoint pattern database heursitics はマンハッタン距離よりも正確なヒューリスティックである。

3.2 Grid Pathfinding

Grid Pathfinding は二次元のグリッドにおいて、障害物を避けつつ初期位置からゴール位置までの経路を求める問題である。本研究では 4 way Grid Pathfinding を扱った。4 way Grid Pathfinding では上下左右の位置にあるグリッドに移動することが出来る。本研究ではグリッドの大きさを 5000×5000、障害物のある確率を 0.45 に設定した。本研究ではゴール距離までのマンハッタン距離をヒューリスティック関数とした。マンハッタン距離は admissible なヒューリスティック関数である。

3.3 Travelling Salesperson Problem (TSP)

Travelling Salesperson Problem (TSP) は、都市の集合とそれぞれの 2 都市間の距離が与えられたとき、全ての都市を一度ずつ訪れ、出発都市に戻る為の最短経路を求める問題である。

TSP で使われるヒューリスティック関数の一つは **Minimum Spanning Tree** である。Minimum Spanning Tree はグラフの全ての頂点を含む、最も辺の長さが小さい部分グラフである。これはゴールまでの経路よりも短いか同等である。なぜなら、ゴールまでの経路の方が短いならば、それが Minimum Spanning Tree になるはずであるからである。よって、Minimum Spanning Tree は admissible なヒューリスティック関数である。

より簡単なヒューリスティックとして **Round Trip Distance** がある。Round Trip Distance は現在の都市から最も離れた未訪問の都市までの距離の 2 倍を返す。このヒューリスティックは Minimum Spanning Tree よりも不正確であるが、非常に素早く計算が出来る。並列化に伴う高速化はノードの展開が速い方が困難である為、Round Trip Distance での高速化の方が難しいと考えられる。本研究は並列化の効率を調べることが目的である為、Round Trip Distance をヒューリスティックに用いる。本研究では正方形内にランダムに都市を配置し、都市間の直線距離をコストとした。

Minimum Spanning Tree、および Round Trip Distance による A*探索は TSP を解くには効率的な方法ではないことに注意されたい。state-of-the-art とされる手法は線形計画法を用いるものである [14]。本研究では HDA*、SafePBNF が木構造の探索空間でどのように振る舞うかを示す為に用いる。

4　並列化に伴う探索オーバーヘッドの定性的な解析

4.1　先行研究における探索オーバーヘッドの解析

Kishimoto らはノードの f 値によって探索ノードの分類を行うことで HDA*の探索オーバーヘッドの分析を行った [2]。c^* を最適解のコストとする。展開したノードのうち、f 値が $f < c^*$ となるノードの割合を $R_<$、$f = c^*$ となるノードの割合を $R_=$、$f > c^*$ となるノードの割合を $R_>$、重複して展開されたノードの割合を R_r とおく。逐次 A*で探索されるノードの f 値は $f < c^*$ または $f = c^*$ である。$f > c^*$ となるノードは展開されない。同じ状態のノードが複数回展開されることもない。対して HDA*を含む並列アルゴリズムは同じノードを複数回展開しうる。また、$f > c^*$ のノードも展開しうる。$f = c^*$ の展開ノード数も A*と異なる。すなわち、$R_=$, $R_>$, R_r の指標は逐次 A*との比較をせずに探索オーバーヘッドの大きさを推定することが出来る。

Kishimoto らは 24 Puzzle ドメインにおいて HDA*の探索オーバーヘッドの殆どが $R_=$, $R_>$ によるものであり、R_r は僅かであると示した。しかしながら、これらの探索オーバーヘッドの原因に対する分析はまだ行われていない。すなわち、$R_=$, $R_>$, R_r となるノードが何故多くなるのかは分析されていない。

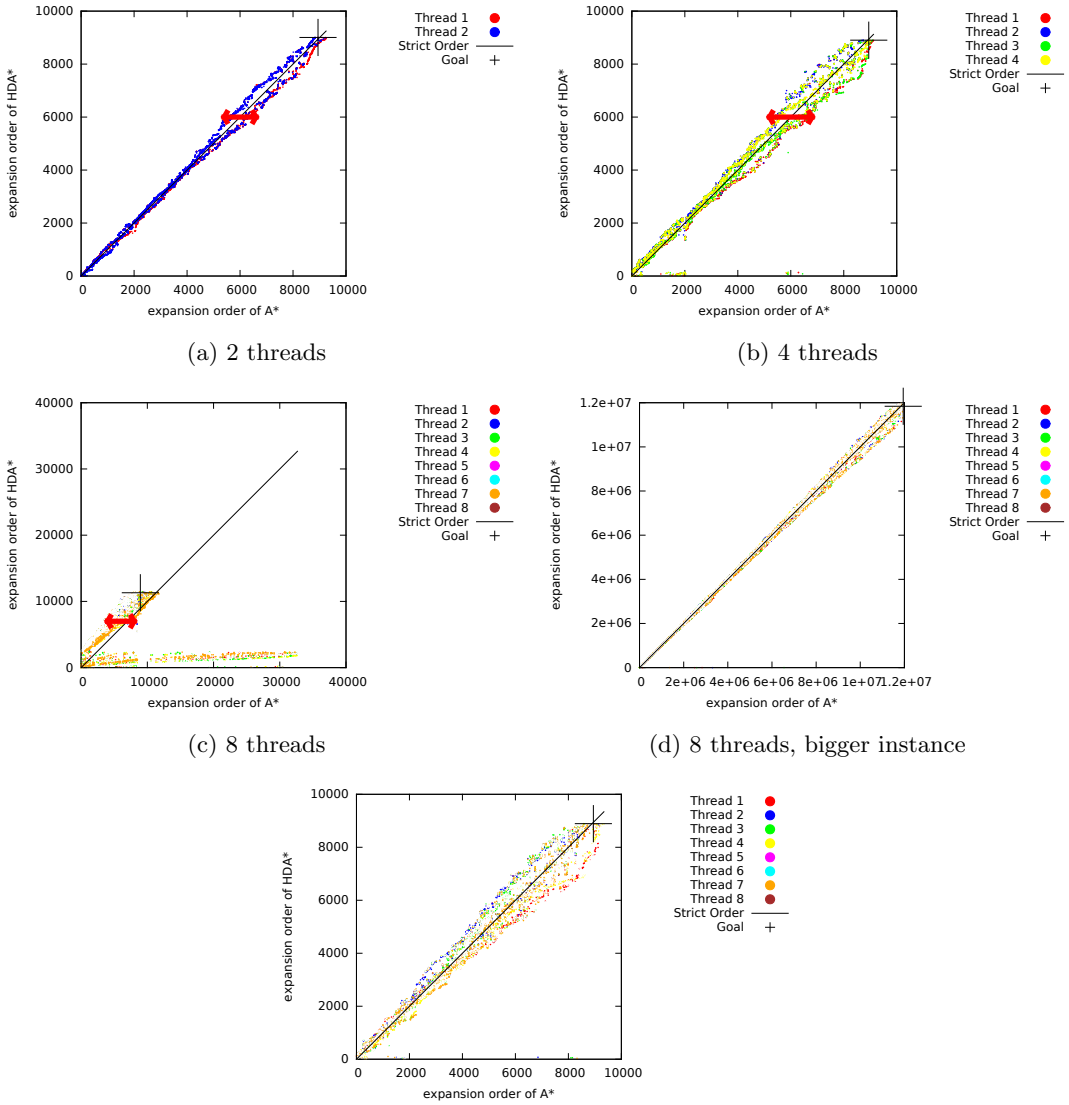
4.2　HDA*のノードの展開順

HDA*の探索オーバーヘッドは、逐次 A*とノードの展開順が異なることが原因であると考えることが出来る。ノードの展開する順番が逐次 A*と完全に一致するならば、探索オーバーヘッドは全く生じない。裏を返せば、逐次 A*と展開順が異なる場合に探索オーバーヘッドが生じると考えられる。よって、HDA*の状態の展開順を逐次 A*のそれと比較をすることで、どのようなパターンで探索オーバーヘッドが生じるのかを分析することが出来ると考えられる。

本実験では 15 Puzzle ドメインを対象に逐次 A*と HDA*においてある状態が展開された順番を記録した。同時に、重複して展開されたノードの数を記録した。インスタンスはランダムに生成した 100 問を対象とした。実験環境は Intel dual-quad Xeon 2.33GHz、32GB RAM、OS は GNU/Linux Ubuntu 14.04 である。本実験では 15 Puzzle ドメインに最適化したコードを利用した。Kishimoto らは MPI message passing library でプロセス間通信を実装したが、本実験では共有メモリ環境において MPI よりも高速な pthread ライブラリを用いた。非同期通信は try_lock で実装した。オープンセットは nested bucket で実装した。なお、オープンセットを heap で実装した場合も概ね同様の結果が得られた。

図 2 は、ある状態が逐次 A*において何番目に展開されたかを横軸に、HDA*において何番目に展開されたかを縦軸に取ってある。直線 $y = x$ を Strict Order と呼ぶ。もし、逐次 A*と HDA*の展開順が完全に一致していた場合、全ての点はこの線の上に並ぶ。図の Goal は最適解が展開された点である。この点が Strict Order よりも上にあれば、その差だけ探索オーバーヘッドが生じているということである。

HDA*のノードの展開順は大きな流れとしては逐次 A*のそれと一致するようである。しかしながら「幅」が存在し、その範囲内で展開順が前後する。図 2a は 2 スレッドでの実行である。例えば、HDA*が 5000 番目周辺に展開するノードは、逐次 A*だと 4500〜5500 番目に展開されているノードである。すなわち、展開順におおよそ 1000 ノード分の順不同が生じている。ここで生じる HDA*と逐次 A*の展開順が前後する範囲を**バンド**、バンドが生じる現象を**バンド効果**と定義する。図 2a, 2b, 2c で示している赤い両矢印はこのバンド



(a) 2 threads

(b) 4 threads

(c) 8 threads

(d) 8 threads, bigger instance

(e) delayed expansion

図 2: HDA*と逐次 A*のノードの展開順の比較

を図示したものである。バンドはスレッド数が大きい程大きい傾向にある。バンドは実行時間を通して大きくなっていくが、展開ノード数が増加する割合に対して小さい。図 2d はより難しいインスタンスでの実験である。このように、展開ノード数の大きな難しい問題程、展開ノード数に対する相対的なバンドの大きさは小さくなると考えられる。これは 4.4 節でも実験的に実証する。

特にスレッドの数が大きい場合に、探索の冒頭で展開順が逐次 A*から大きく外れる。この現象を**バースト効果**と定義する。バースト効果は初期化の終わっていないスレッドがある時間帯に起こる。初期化の終わっていないスレッドがある場合、それらのスレッドの担当するノードを迂回した、限られた探索空間の中で展開を進めることになる。探索空間が異なるので、展開順も逐次 A*から外れてしまうと考えられる。また、バースト効果で展開されたノードは殆どが重複したノードになる。限られた探索空間内を迂回して展開したノードで

あるので、殆どのノードは最短ではない経路で生成される為である。ノードの展開が遅い場合、スレッドの初期化にかかる時間は相対的に小さくなる。バースト効果はノードの展開が非常に速い場合にのみ生じると考えられる。図 2e はノードの展開に無関係の計算[*1]を挿入し、展開速度を遅くした場合の実行結果である。無関係の計算を挿入しない場合の展開速度はおよそ 10^7 ノード/秒 (100 問の平均値) であるのに対して、図 2e の実行の展開速度はおよそ 10^4 ノード/秒である。

表 1: 平均の R_r

スレッド数	2	4	8
R_r	8.97e-07	2.83e-05	2.61e-05

表 1 は全インスタンスの平均のノードの重複率, R_r である。改めて、R_r は以下のように定義される。

$$R_r = \frac{\text{再展開したノードの数}}{\text{展開したノードの数}} \tag{4}$$

HDA*は辺のコストが均一なドメインにおいてほぼノードの重複がないことが知られている [2]。本実験においてもノードの重複は殆どなかった。HDA*は展開済のノードの全てをクローズドセットに保存する。殆どの場合において同じノードを再展開することはない。HDA*において重複が生じる場合は、ある状態に対して複数の経路があり、より長い経路が先に探索された場合のみである。Kobayashi らがノードの重複が問題となるシーケンスアライメント問題において、ノードの重複による探索オーバーヘッドついて議論しているのでこちらを参照されたい [15]。

これらの結果はそれぞれ 1 インスタンスのものを代表的に示したが、実験対象とした 100 問のどのインスタンスに対しても同様のパターンを確認することが出来た。

4.3 探索オーバーヘッドの分類

前節の結果から、HDA*の探索オーバーヘッドの原因を 3 つに分類した。

1. バンド効果

   バンド効果は HDA*と逐次 A*の展開順が前後する為に生じる探索オーバーヘッドである。逐次 A*と展開順が前後するので、実行時間の最後に、逐次 A*であれば展開しないノードを展開してしまう。ここで余分に展開するノードの数はバンドの大きさを上限とする。バンドの大きさは問題サイズが大きいほど相対的に小さくなるので、バンド効果による探索オーバーヘッドは問題サイズが大きくなるほど小さくなる。なお、バンド効果によって逐次 A*よりも最適解の展開順が速くなることがある。$f < c^*$ のノードが全て展開されていれば、その時点で探索を終了することが出来る。

2. バースト効果

[*1] 各展開毎に入力の整数を二乗し 999943 の剰余を求める計算を 100,000 回繰り返した。計算結果は次の入力となる。

バースト効果はスレッドの初期化にかかる時間の為に生じる探索オーバーヘッドである。バースト効果は探索の冒頭にしか生じない為、問題サイズが大きいほど、バースト効果による探索オーバーヘッドの割合は小さくなる。この現象はノードの展開速度が速い場合にのみ起こると考えられる。

3. ノードの重複

ノードの重複は同じノードを複数回展開することによる探索オーバーヘッドである。HDA*は展開済ノードの全てをクローズドセットに保存するので、ノードの重複は少ない。重複が生じる場合は、ある状態に対して複数の経路があり、より長い経路が先に探索された場合のみである。

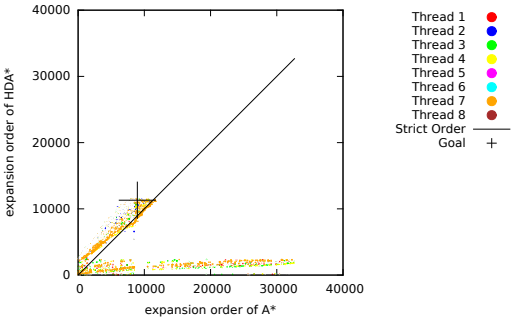


図 3: HDA* with simple hash

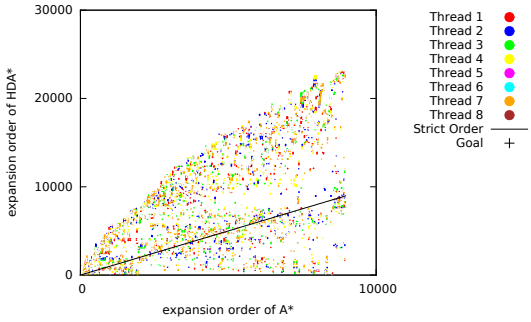


図 4: SafePBNF

上述の探索オーバーヘッドの分類はハッシュ関数が十分な性能を持つ場合のみ有効であると考えられる。図 3 は Burns ら HDA*の性能評価に用いた Simple Hash による HDA*の展開順である。Simple Hash はクローズドセットの為のハッシュ関数をそのまま利用したものである。本実験のクローズドセットのハッシュ関数は Korf と Schultze による Perfect Hashing を用いている [16]。この Perfect Hashing は全ての状態に対して固有のハッシュ値を返す。すなわち、状態とハッシュ値が全単射に対応する。

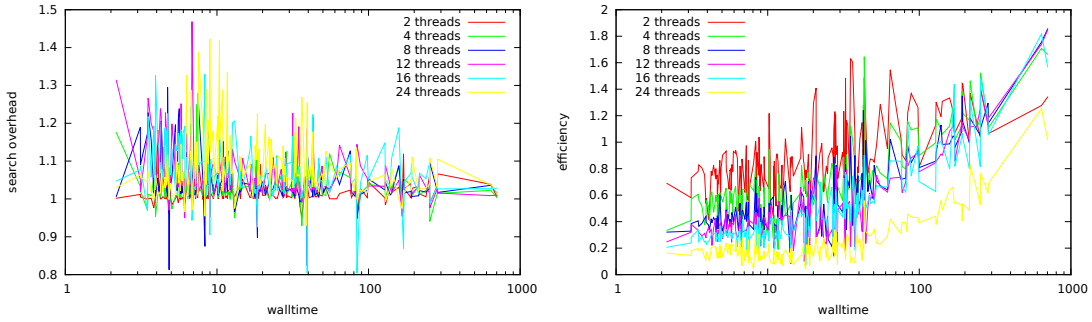
この Simple Hash による分配はうまくいっていないようである。一つのハッシュ値が一つの状態に対応するので、これをスレッドの数で割れば状態空間を均等に分割出来る。しかしながら、探索で現れる状態は 15 Puzzle ドメインの状態空間に対して非常に小さく、偏りのある部分集合である。Simple Hash は状態空間を均等に分割するが、不偏的に均等ではないと考えられる。一方、Zobrist Hash はノードの各状態の情報を利用してハッシュ値を計算する為、より不偏的に均等な分配が可能であると考えられる。Zobrist Hash と Simple Hash の詳細な性能比較は 5 章で行う。

図 4 は SafePBNF の展開順である。この図のみ、縦軸と横軸のスケールが異なることに注意されたい。逐次 A*の展開順とは大きく異なる順番でノードを展開していることが伺える。

バンド効果, バースト効果は問題サイズが大きくなるに従い相対的に小さくなる。ノードの重複は問題となる大きさではない。以上の分析から、Zobrist Hash を用いた HDA*の探索オーバーヘッドは問題サイズが大きくなる程小さくなると推測される。次節にてこれを実験的に検証する。

4.4　問題サイズに対する HDA*の探索オーバーヘッドと実行時間の変化

前節の分析にて、問題サイズが大きくなるほど探索オーバーヘッドが小さくなるという仮説を議論した。本節ではこの仮説を実験的に検証する。15 Puzzle を対象に、問題毎の展開ノード数と高速化効率の分析を行った。15 Puzzle のインスタンスはランダムに生成した 200 問を対象にした。これらの問題は逐次 A*で 1〜1000 秒の実行時間で解ける問題である。実験環境は Intel Xeon E5-2670 2.33GHz (24 core), 64GB RAM, OS は GNU/Linux Ubuntu 14.04 である。なお、前節の実験環境である Intel dual-quad Xeon 2.33GHz, 32GB RAM でも実験を行い、概ね同様の結果が得られた。



(a) Search Overhead　　(b) Efficiency

図 5: 15 Puzzle

図 5a は横軸にそのインスタンスの逐次 A*における実行時間、縦軸に HDA*の探索オーバーヘッドを取っている。大きな傾向として、問題サイズが大きい程探索オーバーヘッドが 1 に近づくという結果を得た。図 5b は横軸にそのインスタンスの逐次 A*における実行時間、縦軸に HDA*の高速化効率を取っている。どのスレッド数でも、問題サイズが大きい程高速化効率が大きくなる傾向が見られる。これは探索オーバーヘッドが改善される傾向に呼応すると考えられる。スレッド数毎に見ると、スレッド数が多くなる程に高速化効率を発揮する為に必要な時間が大きくなる。特に 24 スレッドの実行は非常に大きなインスタンスを必要とする。

興味深い現象として、特に大きな問題インスタンスに対してしばしば superlinear speedup が生じる。多くのインスタンスにおいて発生している為、偶然的な要素だけでは説明されないと考えられる。逐次 A*と 1 スレッドの HDA*の実行時間はほぼ同じであるので、本実験の逐次 A*の実装に問題があるとは考えにくい。また、Kishimoto らの分散メモリ環境における 24 Puzzle の実験においても、一部のインスタンスで superlinear speedup が生じている [2]。superlinear speedup が生じる原因は調査中であるが、クローズドセットのキャッシュ効率の向上が原因ではないかと推測している。ノードは Zobrist Hash によりスレッドに分配され、そこで更にハッシュ関数によりクローズドセットに割り当てられる。これによって各スレッドがアクセスするクローズドセットの大きさが小さくなり、キャッシュ効率が向上すると考えられる。この仮説の検証は今後の研究課題である。

アルゴリズムの性能評価は平均の実行時間や展開ノード数を論じることが殆どである。これは偶然的な要素を取り除く為である。しかしながら、並列探索における高速化効率は問題サイズに対して依存する。然るに本質的な高速化効率への知見を得るためには、平均化した高速化効率による議論ではなく、問題サイズ毎に検討

をする必要がある。その為、次章の議論において実行時間と探索オーバーヘッドは平均値ではなく、問題インスタンス毎に示す。

5　HDA*と SafePBNF の性能比較実験

5.1　先行研究における比較実験の問題点

Burns らの比較実験には複数問題点がある [3]。まず、HDA*のハッシュ関数を Kishimoto らは Zobrist Hash を用いて実装したのに対して、Burns らは Simple Hash によって実装している。Zobrist Hash による分配が Simple Hash のそれよりも大きく優れることは 4.3 節で論じた。

また、Burns らの実験では HDA*, SafePBNF のオープンセットを heap で実装している。コストが整数のドメインはオープンセットを nested bucket で実装することが出来る。heap に対するアクセスが $O(\log n)$ であるのに対して nested bucket へのアクセスは $O(1)$ である。Burns らは別の論文で、15 Puzzle ドメインを対象に逐次 A*の最適化を議論しており、heap 実装と nested bucket 実装の比較実験を行い、nested bucket 実装が 3 倍程度高速であることを示した [17]。HDA*と SafePBNF は複数のオープンセットがある。HDA*はスレッドの数に分割され、SafePBNF は nblock の数に分割される。nblock は 15 Puzzle の場合 3360 個に分割される。よって、両アルゴリズムの nested bucket 実装による高速化の割合は異なると考えられるが、まだ検証はされていない。

もう一つ重要な問題点として、ベンチマーク問題のサイズが小さいということが挙げられる。4.4 節で議論したように、HDA*の高速化効率は 1000 秒程度の問題まで向上し続ける。しかしながら Burns らの用いた問題集は逐次 A*で 10 秒未満のものが殆どである。よって、ベンチマーク問題の問題サイズを考慮した比較・解析が必要である。

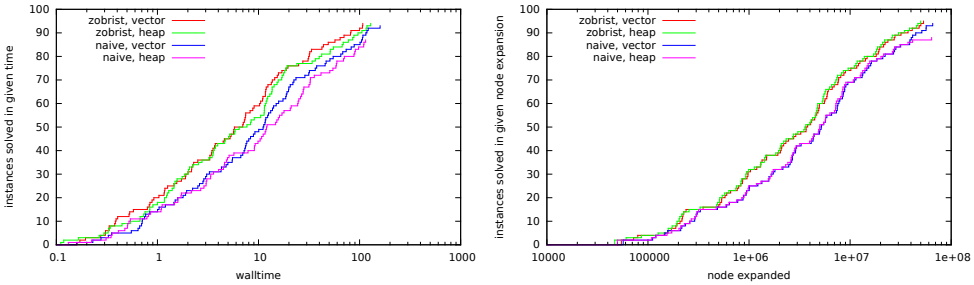
実験マシンは Intel dual-quad Xeon 2.33GHz プロセッサ, 32GB RAM のものを使用した。OS は GNU/Linux Ubuntu 14.04 である。スレッドは pthread ライブラリで実装した。なお、本実験は様々な問題ドメインとアルゴリズムを比較するために、4 章とは異なる実装のコードを利用した。

5.2　HDA*のチューニング

本節では HDA*のハッシュ関数とオープンセットの実装の与える影響を実験的に示す。

4.3 節にて、Simple Hash は分配に問題があると分かった。Simple Hash と Zobrist Hash による実装の実行時間と展開ノード数を比較した。本実験では以上の 2 つの実装の違いが HDA*の実行時間・探索オーバーヘッドに与える影響を比較する。インスタンスは Korf が用いた 100 問を用いた [8]。並列アルゴリズムは全て 8 スレッドの実行による。実行時間の上限は 2000 秒とした。

図 6a は横軸に実行時間、縦軸にその時間内に解けた問題の数を取っている。どの問題サイズにおいても、Zobrist Hash と nested bucket による実装が一番速いと結論出来る。特に Zobrist Hash を用いた実装は Simple Hash による実装よりも 2 倍程度速い。図 6b は横軸を展開ノード数にしたものである。図によると、Zobrist Hash は Simple Hash よりも展開ノード数が小さい。Simple Hash はどの問題サイズでもおおよそ Zobrist Hash の 2 倍の展開ノード数になっている。よって Zobrist Hash の実装の方が高速であるのは展開ノード数の違いによるものであると結論出来る。Zobrist Hash が Simple Hash よりも大きく性能が優れるという結果は 4.3 節における分析結果とも一致する。

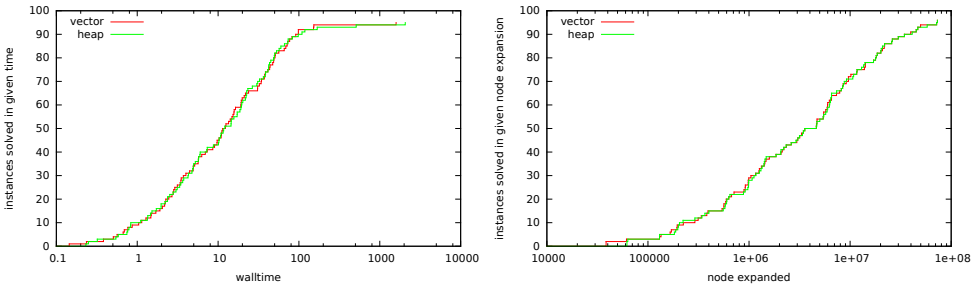


(a) walltime (b) search overhead

図 6: Tuning HDA*

5.3 SafePBNF のチューニング

HDA*において nested bucket によるオープンセットの実装は heap による実装よりも速いことが分かった。SafePBNF においても高速化が期待される。SafePBNF においても Korf の 100 puzzle を対象に比較を行った。SafePBNF のパラメータとして、nblock の個数と最小探索ノード数がある。Burns らは Sliding-tile puzzle 問題において、nblock への分割方法が two-tile projection であり、最小探索ノード数が 8 である場合に最適であると示した。two-tile projection は、ブランク、tile-1、tile-2 の位置が同じノードを一つの nblock とする分割方法である。このとき nblock は 3360 個になる。Burns らの実験は dual quad-core Intel Xeon E5320 1.86GHz プロセッサ、16 GB RAM で行われた。本実験の実験環境と類似している為、Burns らの提案したパラメータを用いた。並列アルゴリズムは全て 8 スレッドの実行による。実行時間の上限は 2000 秒とした。



(a) walltime (b) search overhead

図 7: Tuning SafePBNF

図 7 が実行結果である。SafePBNF においては、HDA*ほどの高速化が得られないようである。heap に対するアクセスは $O(\log n)$ であるのに対して nested bucket は $O(1)$ である。本実験で SafePBNF は 3360 個のオープンセットに分割されている。その為各オープンセットが小さく、$O(\log n)$ と $O(1)$ の差が小さい為、nested bucket 実装による高速化が小さいと考えられる。翻って、SafePBNF は heap 実装による遅延の影響が小さいとも言える。

5.4 HDA*と SafePBNF の性能比較実験

5.2 節より、HDA*は Zobrist Hashing と nested bucket による実装が最も速いと結論された。一方 5.3 節では、SafePBNF は nested bucket と heap による実装で大きなパフォーマンスの違いは見られなかった。一方、heap は nested bucket よりも広汎なドメインで実装をすることが出来るという利点がある。よって、nested bucket と heap の両方の実装の性能評価をすることは有意義である。

以下、15 Puzzle, 24 Puzzle, Grid Pathfinding, TSP で実験を行った。15 Puzzle は nested bucket と heap のそれぞれ、24 Puzzle、 Grid Pathfinding は nested bucket, TSP は heap で実装をした。TSP のヒューリスティックは Round Trip Distance、都市の数は 13 とした。HDA*のハッシュ関数は Zobrist Hash による。

SafePBNF の nblock への分割方法と nblock の数はそれぞれ以下である。15 Puzzle は two-tile projection (nblock=3360), 24 Puzzle も同様に two-tile projection (nblock=13820) による。Grid Pathfinding は 100×100 のグリッドを一つの nblock とした (nblock=2500)。TSP は最初に訪れた 3 つの都市による (nblock=1716)。 最小探索ノード数は全て 8 とした。

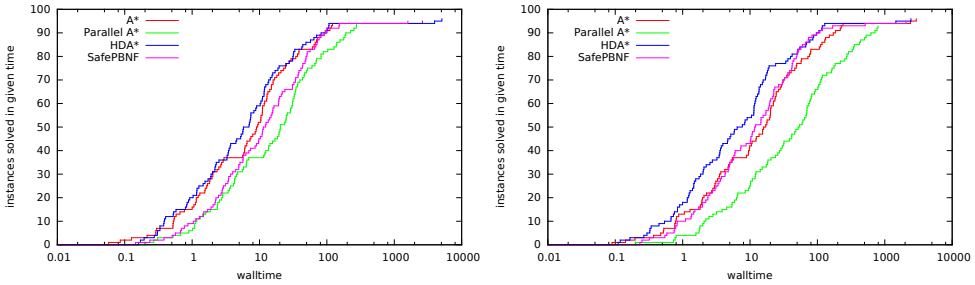
並列アルゴリズムは全て 8 スレッドの実行による。15 Puzzle, 24 Puzzle の実行時間の上限は 6000 秒、他のドメインの実行時間の上限は 2000 秒とした。

図 8 が実行時間の比較である。それぞれの図は、横軸に実行時間、縦軸にその実行時間内に解ける問題の数を示してある。これらの結果から、特に実行時間の大きい問題ドメイン・インスタンスにおいて、HDA*は SafePBNF よりもパフォーマンスに優れると結論出来る。一方、実行時間が 5 秒未満程度の小さい問題に対しては SafePBNF の方が優れる場合が多い。4 章で議論したように、HDA*は実行時間の冒頭と末尾に探索オーバーヘッドがある。実行時間が短い場合、これらのオーバーヘッドの割合が大きくなる。よって、小さな問題インスタンスに対しては探索オーバーヘッドが大きく、SafePBNF の方が速いと考えられる。特に Grid Pathfinding の 1 秒程度の問題において HDA*は大きな探索オーバーヘッドを持つ。

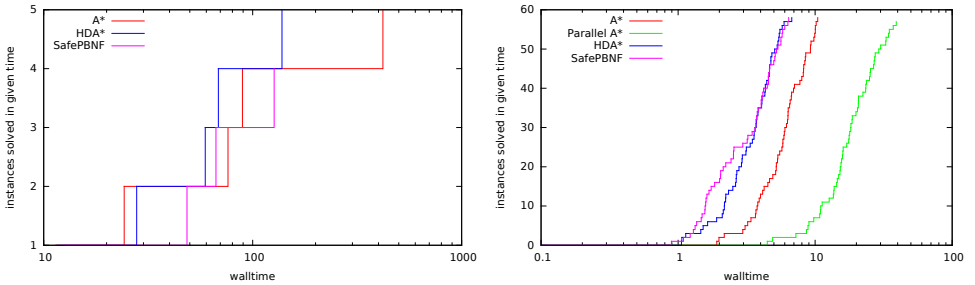
図 9 は展開ノード数の比較である。それぞれの図は、横軸に展開ノード数、縦軸にその展開ノード数以内に解ける問題の数を示してある。実行時間の短い Grid Pathfinding を除き、HDA* vs. SafePBNF の展開ノード数はあまり変わらない。4 章の結論とこの結果を踏まえると、HDA*は十分に大きな問題に対しては探索オーバーヘッドの殆どない手法であると結論出来るだろう。Burns らが HDA*の探索オーバーヘッドを指摘したのはハッシュ関数を Simple Hash で実装したからである。図 3 及び図 6 より、Simple Hash は明らかに性能が悪い。その原因は調査中であるが、クローズドセットに対してハッシュ値が偏っている可能性があると推測している。

図 8b は 15 Puzzle ドメインでオープンセットを nested bucket で実装した場合の実行時間の比較である。このドメインはノードの展開が非常に高速なドメインである。逐次 A*で 10^5〜10^6 ノード/秒が展開される。この場合でも HDA*は逐次 A*に対して高速化が可能である。これは 4 章の実験結果と一致する。一方、SafePBNF は逐次 A*に対して高速化が見られない。図 8b は 15 Puzzle ドメインでオープンセットを heap で実装した場合の実行時間の比較である。heap 実装の場合、HDA*と SafePBNF は共に逐次 A*よりも高速である。加えて、この実験結果は heap 実装においても HDA*が SafePBNF よりもパフォーマンスに優れるということを示す。

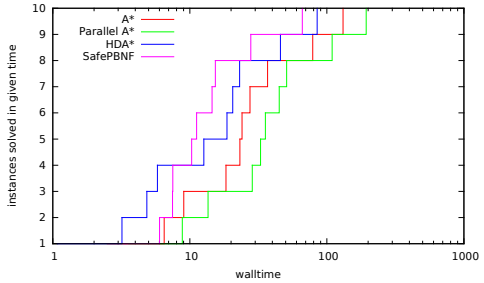
nested bucket 実装で SafePBNF が速くならない原因は二つの理由が考えられる。一つは、nested bucket 実装によってノードの展開が高速化したことである。ノードの展開が高速であると、並列化のオーバーヘッド



(a) 15 Puzzle (nested bucket)　(b) 15 Puzzle (heap)



(c) 24 Puzzle (nested bucket)　(d) Grid Pathfinding (nested bucket)



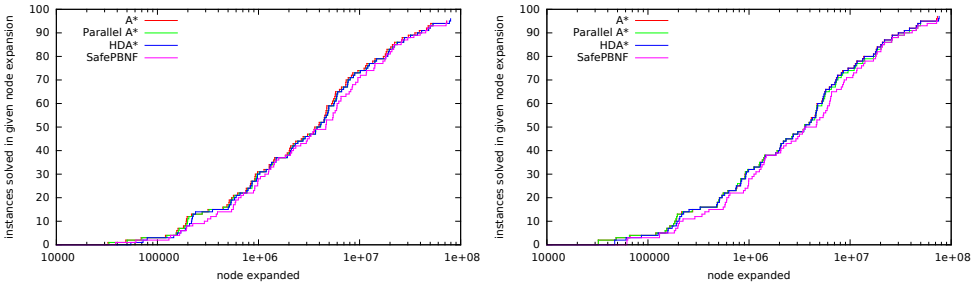
(e) TSP (heap)

図 8: HDA*と SafePBNF の実行時間の比較

による影響が比較して大きくなる。 もう一つに、そもそも SafePBNF の高速化は主に heap を分割することによるものであった可能性がある。SafePBNF は 15 Puzzle の場合 3360 個の nblock に分割する。よって、オープンセットは 1/3360 の大きさであると期待される。この時の heap へのアクセス速度の違いは非常に大きいと考えられる。Burns らは SafePBNF (3360 nblocks) と HDA* (8 threads) の heap へのアクセスにかかる CPU 時間の計測を行い、SafePBNF のそれがほぼ 0 秒であるのに対して HDA*はおおよそ 0.00001 秒 (中央値) かかることを示している [3]。これは 1 回のアクセスにかかる時間なので、実行時間を通しての合計は非常に大きい。

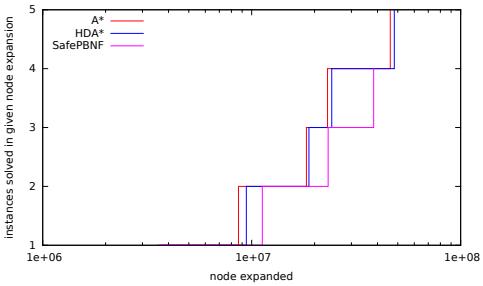
HDA*でも各スレッドの heap を分割することは可能である。ハッシュ関数による仕事の分配を各スレッド内の heap に自然に流用することが出来る。よって、HDA*でも分割 heap を用いることで実数コストドメインにおけるパフォーマンスを向上させることが出来ると考えられる。これの実験的な検証は今後の研究課題である。

SafePBNF は TSP ドメインにおいて効率的である。その理由としては以下の二点が考えられる。一つは前

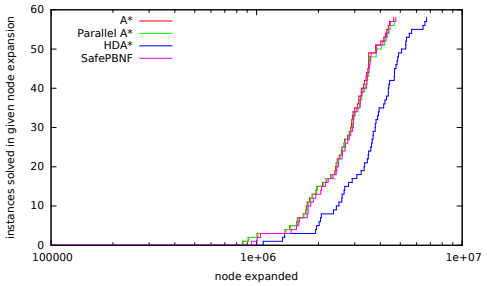


(a) 15 Puzzle (nested bucket)

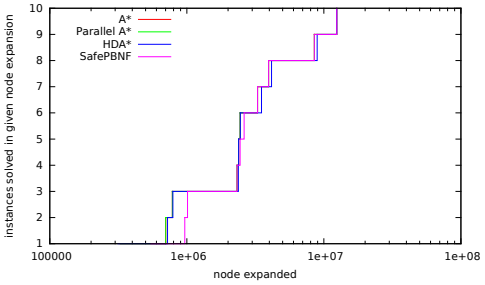
(b) 15 Puzzle (heap)



(c) 24 Puzzle (nested bucket)



(d) Grid Pathfinding (nested bucket)



(e) TSP (heap)

図 9: HDA*と SafePBNF の展開ノード数の比較

述の heap を分割することによる高速化である。もう一つはドメインに解が無数にあるからである。表 2 は HDA*と SafePBNF において、解を発見した平均の回数である。SafePBNF の探索方法は Anytime Search のような特徴を持つ [3]。すなわち、SafePBNF は最適でない解を素早く見つけ、時間と共により良い解を発見することが出来る。最適でない解を発見した場合、その解のコストよりもコストの大きいノードを生成する必要がなくなる。ノードの生成は探索時間の大きな割合を占める為、生成の回数を減らすことは大きな高速化につながる。TSP は深さが都市の数と同じノードは全て解である。都市が 13 個の場合は 12! 通りの解が存在する。SafePBNF は解が無数にある TSP では素早く解を発見でき、その為最適解も高速に見つけられると考えられる。

どのドメインにおいても Parallel A*は逐次 A*よりも遅くなってしまう。この結果は Burns らの実験と一致する [3]。Parallel A*はオープンセットとクローズドセットにロックを必要とする。スレッドはロックが解放されるのを待たなければならない為、大きな同期オーバーヘッドが生じる。なお、Parallel A*は 24 Puzzle ドメインの問題を 1 問も解けなかった為実行結果を省略してある。

表 2: 解の発見回数と最初の解の発見時間の平均

	HDA*		SafePBNF	
	解の発見回数	最初の解の発見時間	解の発見回数	最初の解の発見時間
15 Puzzle (nested bucket)	1	14.56	1.14	20.73
24 Puzzle	1	59.82	1	67.04
Grid Pathfinding	1.02	3.45	1.02	3.30
TSP13	1	16.80	**2.60**	**8.56**

6 おわりに

本研究の貢献は大きく分けて 2 つある。

1. HDA*の探索オーバーヘッドの定性的な分析

   HDA*の探索オーバーヘッドの原因を三つに分類を行い、十分問題サイズが大きい場合に探索オーバーヘッドが無視出来る程度に小さいことを定性的に示した。また、探索オーバーヘッドと高速化効率を問題インスタンス毎に示すことで、それを実験的にも実証した。同時に、十分問題サイズが大きい場合に高速化効率が perfect linear speedup に近づくことを示した。本研究は共有メモリ環境において行われたが、分散メモリ環境においても同様の結果が得られると推測している。その実験的な実証は今後の研究課題である。並列探索における探索オーバーヘッドは定量的な解析は一般的であるが、定性的な解析は我々の知る限りない。本研究で得られた知見は広く並列探索一般に適応されると考えられる。加えて、本研究は新しい探索オーバーヘッドの分析手法を示した。これは HDA*、SafePBNF に限らず他の並列探索においても有効な分析方法であるだろう。

2. HDA*と PBNF の再評価

   先行研究の分析の問題点を指摘し、5 つのドメインにおいて HDA*と SafePBNF の比較実験を行った。Sliding-tile ドメインにおいて、マルチコア環境においても HDA*の方が SafePBNF よりもパフォーマンスに優れることを実験的に示した。同時に、SafePBNF と比較しても HDA*は探索オーバーヘッドが大きくないことを示した。ドメイン間での性能の違いから、SafePBNF の高速化が heap 実装に依存する可能性があることを指摘した。HDA*にも分割 heap を実装することで同様の高速化が期待される。この検証は今後の研究課題である。

7　謝辞

研究室の先輩方にはミーティングや中間発表などで沢山の有益なアドバイスを頂きました。特に同期の堀江君とは本当に毎日議論を重ね、沢山のことを学ばせて頂きました。指導教員である福永先生には、初めて情報科学に触る私に対してとても丁寧にご指導して頂きました。研究とは、情報科学とは、プログラミングとは、あらゆることを教えて頂きました。何よりも、研究が楽しいということを教えて頂いたことに深く感謝致します。修士課程でも、一つでも多くのことを学ばせて頂きたいと思います。

参考文献

[1] Peter E Hart and J Nils. Formal Basis for the Heuristic Determination. *IEEE Transactions on Systems Science and Cybernetics*, Vol. 4, No. 2, pp. 100–107, 1968.

[2] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, Vol. 195, pp. 222–248, 2013.

[3] Ethan Andrew Burns and Sofia Lemons. Best-First Heuristic Search for Multicore Machines. *Journal of Artificial Intelligence Research*, Vol. 39, No. 1, pp. 689–743, 2010.

[4] Vipin Kumar, K Ramesh, and V Nageshwara Rao. Parallel Best-First Search of State-Space Graphs: A Summary of Results. Vol. 88, pp. 122–127. Citeseer, 1988.

[5] Matthew Evett, James Hendler, Ambuj Mahanti, and Dana Nau. PRA*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, Vol. 25, No. 2, pp. 133–143, 1995.

[6] Nihar R Mahapatra and Shantanu Dutt. Scalable global and local hashing strategies for duplicate pruning in parallel A* graph search. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 7, pp. 738–756, 1997.

[7] John W Romein, Aske Plaat, Henri E Bal, and Jonathan Schaeffer. Transposition table driven work scheduling in distributed search. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI)*, pp. 725–731, 1999.

[8] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, Vol. 27, No. 1, pp. 97–109, 1985.

[9] A L Zobrist. A new hashing method with application for game playing. *reprinted in International Computer Chess Association Journal*, Vol. 13, No. 2, pp. 69–73, 1970.

[10] Rong Zhou and Eric A Hansen. Parallel structured duplicate detection. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI)*, Vol. 22, pp. 1217–1223, 2007.

[11] Rong Zhou and Eric A Hansen. Structured duplicate detection in external-memory graph search. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI)*, pp. 683–689, 2004.

[12] Richard E. Korf and Ariel Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, Vol. 134, No. 1, pp. 9–22, 2002.

[13] Joseph C Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, Vol. 14, No. 3, pp. 318–334, 1998.

[14] David L Applegate. *The traveling salesman problem: a computational study.* Princeton University Press, 2006.

[15] Yoshikazu Kobayashi, Akihiro Kishimoto, and Osamu Watanabe. Evaluations of hash distributed A* in optimal sequence alignment. Vol. 22, pp. 584–590, 2011.

[16] Richard E Korf and Peter Schultze. Large-scale parallel breadth-first search. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI)*, Vol. 5, pp. 1380–1385, 2005.

[17] Ethan Andrew Burns, Matthew Hatem, Michael J Leighton, and Wheeler Ruml. Implementing fast heuristic search code. 2012.

索引