# 大規模並列環境におけるグラフ最良優先探索の効率的な仕事分配手法

陣内 佑
東京大学大学院 総合文化研究科

## 1 はじめに

本報告は平成 28 年度スーパーコンピューター若手・女性利用者推進制度の課題「大規模並列環境におけるグラフ最良優先探索の効率的な仕事分配手法」の成果を報告する。内容は Journal of Artificial Intelligence に採択された論文 [11] の日本語訳・要旨である。

## 2 Introduction

A*探索 [4] はグラフ上の最短経路を求めるアルゴリズムであり、人工知能分野における多くの応用（自動行動計画、ロボットの動作計画等）、最適シーケンスアライメント問題、経路探索問題をはじめとする多くの問題を解くのに応用されている。A*探索は実行時間と空間量の両方がボトルネックになるアルゴリズムである。A*探索の並列化は高速化だけでなく分散環境にある複数ノードのメモリを使うことによってメモリ消費量の問題も改善することができる。よって、分散環境において効率的な並列 A*探索を考案することは重要な課題である。

A*探索の並列化手法は数多く提案されている。Hash Distributed A* (HDA*) は分散環境において state-of-the-art とされる手法である [16]。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*のパフォーマンスはノードの分配に使用するハッシュ関数の性能に依存する。岸本ら ([15, 16]) は Zobrist hashing ([19]) を使うことで HDA*が良いロードバランスを達成できることを示した。Burns et al.([2]) は Zobrist hashing による仕事分配は大きな通信オーバーヘッドを生むことを指摘し、abstraction [18] を用いた AHDA*を提案した。AHDA*は通信オーバーヘッドを抑えることができるが、一方でロードバランスが悪い。Abstract Zobrist hashing (AZH) [9] は Zobrist hashing と abstraction の両手法を組み合わせることで良い通信・探索オーバーヘッドを達成する。

これらの手法は通信・探索オーバーヘッドを抑えることを目指しているが、これらのオーバーヘッドが最終的に高速化効率にどのような影響を与えるかについての議論はなされていなかった。そのため仕事分配手法の特定の問題を解決することによってアドホックに高速化を得ていた。本研究ではまず、トップダウンに、並列 A*探索で生じるオーバーヘッドを最小化するための目的関数を提案する。これを踏まえ、与えられた目的関数を近似的に最適化する仕事分配手法を生成する並列探索アルゴリズム GRAZHDA*を提案する。

## 3 背景

### 3.1 並列化オーバーヘッド (Parallel Overheads)

理想的には $n$ プロセスで並列化したら $n$ 倍速くなってほしい。逐次アルゴリズムと比較して、プロセス数倍の高速化が得られることを *perfect linear speedup* と呼ぶとする。しかしながら、並列化にさいして様々なオーバーヘッドがかかるため、殆どの場合 perfect linear speedup は得られない。探索アルゴリズムにおける並列化オーバーヘッドは主に以下の 3 つに分けられる。

**通信オーベーヘッド (Communication overhead, CO)**: 通信オーバーヘッドはプロセス間で情報交換を行うことにかかるオーバーヘッドである。通信する情報は様々なものが考えられるが、オーバーヘッドとなるものはノードの生成回数に比例した回数通信を必要とするものである。すなわち、ノードの生成回数 $n$ に対して $log(n)$ 回しか通信を行わない場合、その通信によるオーバーヘッドは無視出来るだろう。ここではノードの生成回数に対するメッセージ送信の割合を CO と定義する：

$$CO := \frac{\text{\# messages sent to other threads}}{\text{\# nodes generated}}. \tag{1}$$

例えば、ハッシュなどによってプロセス間でノードの送受信を行いロードバランスを行う手法の場合、通信するメッセージは主にノードである。この場合：

$$CO := \frac{\text{\# nodes sent to other threads}}{\text{\# nodes generated}}. \tag{2}$$

となる。CO は通信にかかる遅延だけでなく、メッセージキューなどのデータ構造の操作にかかるコストも含む。CO は特にノードの展開速度が速いドメインにおいて重要なオーバーヘッドになる。一般に、プロセス数が多いほど CO は大きくなる。

**探索オーバーヘッド (Search Overhead, SO):** 一般に並列探索は逐次探索より多くのノードを展開することになる。このとき、余分に展開したノードは逐次と比較して増えた仕事量だと言える。本書では以下のように探索オーバーヘッドを定義する：

$$SO := \frac{\text{\# nodes expanded in parallel}}{\text{\#nodes expanded in sequential search}} - 1. \tag{3}$$

SO はロードバランス (load balance, LB) が悪い場合に生じることが多い。

$$LB := \frac{\text{Maximum number of nodes assigned to a thread}}{\text{Average number of nodes assigned to a thread}}. \tag{4}$$

ロードバランスが悪いと、ノードが集中しているスレッドがボトルネックとなり、他のスレッドはノードがなくなるか、あるいはより f 値の大きい (有望でない) ノードを展開することになり、探索オーバーヘッドになる。探索オーバーヘッドは実行時間だけでなく、メモリ消費量にも影響する。A*探索ではノードを生成した分だけ消費するメモリ量も多くなる。分散メモリ環境においてもコア当りの RAM 量は大きくなるわけではないので、探索オーバーヘッドによるメモリ消費は問題となる。

**同期オーバーヘッド (Coordination Overhead)** 同期オーバーヘッドは他のスレッドの処理を待つためにアイドル状態にならなければならない時に生じるオーバーヘッドを指す。アルゴリズム自体が同期を必要としないものだとしても、メモリバスのコンテンションによって同期オーバーヘッドが生じることがある [2, 16].

これらのオーバーヘッドは独立ではなく、むしろ相互に関係しており、トレードオフの関係にある。多くの場合、通信・同期オーバーヘッドと探索オーバーヘッドがトレードオフの関係にあたる。A*探索の並列化に関するより詳細な議論は [3] を参考にされたい。

### 3.2 Hash Distributed A*

Hash Distributed A* (HDA*) [16] は分散環境における state-of-the-art の並列 A*探索アルゴリズムである。HDA*の各プロセスはそれぞれローカルなオープンリスト、クローズドリストを保持し、排他的なアクセスを行う。グローバルなハッシュ関数によって全ての状態は一意に定まる担当のプロセスが定められる。各プロセス $T$ の動作は以下を繰り返す：

1. プロセス $T$ はメッセージキューを確認し、ノードが届いているかを確認する。届いているノードのうち重複でないものをオープンリストに加える (A*同様、クローズドリストに同

じ状態が存在しないか、クローズドリストにある同じ状態のノードよりも $f$ 値が小さい場合に重複でない)。

2. オープンリストにあるノードのうち最もプライオリティ($f$ 値) の高いノードを展開する。生成されたそれぞれのノード $n$ についてハッシュ値 $H(n)$ を計算し、ハッシュ値 $H(n)$ を担当するプロセスに非同期的に送信される。

HDA*の重要な特徴は 3 つある。まず、ハッシュ関数を用いてノードのプロセスへの割当を決定するため、自動行動計画問題などの状態空間に存在するノードが非明示的に定義される問題に対して適用可能である。また、HDA*は非同期通信を行うため、同期オーバーヘッドが非常に小さい。各プロセスがそれぞれローカルにオープン・クローズドリストを保持するため、これらのデータ構造へのアクセスにロックを必要としない。次に、HDA*は手法が非常にシンプルであり、ハッシュ関数 $Hash : S \rightarrow 1..P$ を必要とするだけである ($P$ はプロセス数)。しかしながらハッシュ関数は通信オーバーヘッドとロードバランスの両方を決定する為、その選択はパフォーマンスに非常に大きな影響を与える。

HDA*が提案された論文 [16] では Zobrist hashing [20] がハッシュ関数として用いられていた。状態 $s = (x_1, x_2, ..., x_n)$ に対して Zobrist hashing のハッシュ値 $Z(s)$ は以下のように計算される：

$$Z(s) := R_0[x_0]\ xor\ R_1[x_1]\ xor\ \cdots\ xor\ R_n[x_n] \tag{5}$$

Zobrist hashing は初めにランダムテーブル $R$ を初期化する 2。これを用いてハッシュ値を計算する。

**Algorithm 1:** ZHDA*

**Input:** $s = (x_0, x_1, ..., x_n)$

```
1 hash ← 0;
2 for each x_i ∈ s do
3 |   hash ← hash xor R[x_i];
4 Return hash;
```

**Algorithm 2:** Initialize ZHDA*

**Input:** $V = (dom(x_0), dom(x_1), ...)$

```
1 for each x_i do
2 |   for each t ∈ dom(x_i) do
3 |   |   R_i[t] ← random();
4 Return R = (R_1, R_2, ..., R_n)
```

Zobrist hashing を使うメリットは 2 つある。一つは計算が非常に速いことである、XOR 命令は CPU の演算で最も速いものの一つである。かつ、状態の差分を参照することでハッシュ値を計算することが出来るので、アクション適用によって値が変化した変数の $R[x]$ のみ参照すれば良い。もうひとつは、状態が非常にバランスよく分配され、ロードバランスが良いことである。一方、この手法の問題点は通信オーバーヘッドが大きくなってしまうことにある。この問題を解決するために State abstraction という手法が提案された [2]。State abstraction は状態 $s = (x_1, x_2, ..., x_n)$ に対して簡約化状態 (abstract state) $s' = (x'_1, x'_2, ..., x'_m), m < n, x'_i = x_j (1 \leq j \leq n)$ を返す関数を用い、状態のハッシュ値は対応する abstract 状態に対して割り振られたハッシュ値を用いるという手法である。

### 3.3 Abstract Zobrist Hashing

Abstract Zobrist hashing (AZH) は Zobrist hashing と State abstraction を組み合わせることによって両手法の利点を取り入れた手法である [9]。Zobrist hashing が各状態変数をそのまま使ってハッシュ値を計算するのに対し、AZH は変数をまず abstract 変数に投影し、その abstract 変数を使ってハッシュ値を計算する。

AZH によるハッシュ値 $AZ(s)$ は以下の式で計算される。

$$AZ(s) := R[A(x_0)]\ xor\ R[A(x_1)]\ xor\ \cdots\ xor\ R[A(x_n)] \tag{6}$$

$A$ は変数から abstract 変数への多対一の投影関数である。$R$ はランダムに初期化されたテーブルである。図**??**はスライディングタイル問題における AZH の例である。各タイルがどの位置にあるか ($[0-8]$) が状態変数であり、タイルがどの列にあるか ($[0-2]$) を表す abstract 変数に投影される。この abstract 変数の組み合わせを Zobrist hashing への入力としてハッシュ値が計算される。

AZH は State abstraction と Zobrist hashing の利点を組み合わせた手法である。変数を abstract 変数に投影する（これは State abstraction が状態を abstract 状態に投影することに対応する）ことで通信オーバーヘッドを抑えることができ、abstract 変数に対して Zobrist hashing を用いることでロードバランスが達成される。そのため、AZH は Zobrist hashing よりも CO が少なく、abstraction よりも SO が少ないと考えられる。AZH は [9] においてスライディングタイル問題、経路探索問題、プランニング問題において両手法よりも高速化効率が高いことが実験的に示された。

## 4 並列化効率とグラフ分割問題

### 4.1 並列化効率

HDA*の仕事分配手法における先行研究は CO、SO を抑えるための手法を提案したが、それらがどのように実際の高速化につながるかのモデルは議論されていなかった。この節ではまず仕事分配手法が状態空間グラフの分割としてモデルすることが出来、通信オーバーヘッド・ロードバランスはカットエッジの数・分割のバランスに対応することを示す。

HDA*は解の最適性を保証するためにゴールノードと $f$ 値が $f < f^*$ となるノードをすべて展開する必要がある。これらのノードの集合を $S$ と置く。仕事分配手法は状態空間グラフと同型の無向グラフである仕事グラフ $G_W$ を分割する。仕事グラフはノード集合 $S$ と状態空間グラフ内で $S$ にあるノード間をつなぐエッジを無向にしたエッジの集合による。$p$ 個のプロセスに仕事を分配する場合は $G_W$ の $p$-way 分割としてモデルでき、分割 $S_i$ に属するノードはプロセス $p_i$ に割り当てられるとする。

$G_W$ の分割が与えられれば、*LB* と *CO* は *HDA***を実際にマシンで走らせることなしにグラフの構造から直接求めることができる。すなわちアルゴリズムの効率が実際に走らせることなしに評価することが可能である。

LB は分割のロードバランスに対応し、CO はエッジの総数に対する分割間のエッジの数の割合である。

$$CO = \frac{\sum_i^p \sum_{j>i}^p E(S_i, S_j)}{\sum_i^p \sum_{j\geq i}^p E(S_i, S_j)},\quad LB = \frac{|S_{max}|}{mean|S_i|}. \tag{7}$$

$|S_i|$ は $S_i$ に含まれるノードの数、$E(S_i, S_j)$ は $S_i$ と $S_j$ の間のエッジの数、$|S_{max}|$ は $|S_i|$ の最大値、$mean|S| = \frac{|S|}{p}$ である。

次に SO と LB の関係について考える。並列最良優先探索において LB が悪いと SO が大きくなることは実験的に示されていたが [10]、これらの関係の解析は行われていない。

ノードの重複数が無視できる数であり、各プロセスが単位時間当たりに展開できるノードの数は同じだとする。

HDA*は $S$ に含まれるノードをすべて展開しなければならないので各プロセスは $|S_{max}|$ 個のノードを展開する。結果、プロセス $p_i$ は $S$ に含まれないノードを $|S_{max}| - |S_i|$ 個展開することになる。これらのノードが探索オーバーヘッドに相当する。各プロセスの探索オーバーヘッドの総和が $SO$ となる。

$$\begin{aligned} SO &= \sum_{i}^{p}(|S_{max}| - |S_i|) \\ &= p(LB - 1). \end{aligned} \tag{8}$$

### 4.2 高速化効率とグラフ分割

この節ではCOとSOから計算できる実行時間の効率を測るための指標を示す。まず、$eff_{actual} := \frac{speedup}{\#cores}$ を時間効率と定義する。$speedup = T_n/T_1$ であり、$T_n$ は $N$ 個のコアで実行した場合の実行時間である。我々の最終的な目的はこの $eff_{actual}$ を最大化することである。

**通信効率:** すべてのプロセス間の通信コストが一定であると仮定する。通信効率 $eff_c$ は通信コストによる効率の低下であり、$eff_c = \frac{1}{1+cCO}$ となる。$c = \frac{\text{time for sending a node}}{\text{time for generating a node}}$ である。

**探索効率:** 各プロセスが1ノードを展開するためにかかる時間が一定であり、仕事のないコアは存在しないとすると、$p$ プロセスによるHDA*はA*探索が $n$ ノードを展開する実行時間で $np$ ノードを展開する。探索効率 $eff_s$ は探索オーバーヘッドによる効率の低下であり、$eff_s = \frac{1}{1+SO}$ である。

COとLBを使って時間効率 $eff_{actual}$ を推定することができる。$eff_{actual}$ は通信効率・探索効率の積に比例する値と考える: $eff_{actual} \propto eff_c \cdot eff_s$。CO、SO以外にも実行時間に影響を与えるオーバーヘッドは存在するが (e.g. メモリバスコンテンション) [2]、COとSOが実行時間に影響を与える支配的な要素であると仮定する。推定効率 $eff_{esti} := eff_c \cdot eff_s$ を通信効率と探索効率の積と定義する。推定効率を仕事分配手法の実際の実行時間を推定する指標に用いる。

$$\begin{aligned} eff_{esti} &= eff_c \cdot eff_s = 1/\big(\ (1 + cCO)(1 + SO)\ \big) \\ &= 1/\big(\ (1 + cCO)(1 + p(LB - 1))\ \big) \end{aligned} \tag{9}$$

#### 4.2.1 Experiment: $eff_{esti}$ model vs. actual efficiency

$eff_{esti}$ が実際の高速化効率と相関があるかを確認するため、ドメイン非依存の自動行動計画問題において以下の仕事分配手法の性能の比較をする。

- FAZHDA*: fluency-based filtering (FluencyAFG) による AZHDA* [10].
- GAZHDA*: greedy abstract feature generation (GreedyAFG) による AZHDA* [9].
- OZHDA*: Operator-based Zobrist hashing による HDA* [10].
- DAHDA*: dynamic abstraction size threshold [10] による AHDA* [2].
- ZHDA*: Zobrist hashing による HDA* [16].

Fast Downward [5] をもとにこれらの仕事分配手法を用いたHDA*を実装した。プロセス間通信はMPICH3による。ヒューリスティック関数はmerge&shrinkを用いた [6] (abstraction size=1000)。ベンチマークはIPCベンチマーク問題集より並列化による性能の違いが確認できる十分に難しい問題を選んだ。8-core Intel Xeon E5410 (2.33 GHz)、16GB RAM、1000 Mbps Ethernetによるマシン6台のクラスタで実験を行った。100ノードをまとめて1つのMPIのメッセージとした。

テーブル1は高速化の比較である (1プロセスの実行時間/48プロセスの実行時間)。実際の高速化効率 $eff_{actual}$ は実際の実験結果を用いて計算した。$eff_{esti}$ の計算は以下による。各インスタンスに対して仕事グラフ $G_W$ を生成し (すべての $f \leq f^*$ となるノードとそれらの間のエッジを

列挙する)、式 7-9 をもとに LB、CO、SO、$eff_{esti}$ を計算した。図 1b の推定効率 $eff_{esti}$ と高速化効率 $eff_{actual}$ の比較は両者の強い相関を示している。$eff_{actual} = a \cdot eff_{esti}$ への最小二乗法による相関係数 $a$ は 0.86 であり残差分散は 0.013 である。$a$ が 1 よりも小さい理由としては $eff_{esti}$ は通信・探索オーバーヘッド以外に存在するオーバーヘッド (e.g. メモリバスコンテンション [2]) を加味していないことによると考えられる。

## 5 Sparsest Cut による Abstract 変数の生成

仕事グラフを何らかの指標 (e.g. min-cut, sparsest-cut) を用いながら分割することによってプロセスへの仕事の割当を行う手法は科学計算などで使われる方法である [7, 1]。4.2 節では $eff_{esti}$ が HDA*の性能を推定するための良い指標になることを示した。$eff_{esti}$ を最大化するように仕事グラフ $G_W$ を分割することが出来れば理想的仕事分配手法となる。

*Sparsest-cut* はグラフの *sparsity* を最大化する目的関数である　[17]。ここで sparsity は以下のように定義する。

$$Sparsity := \frac{\prod_i^k |S_i|}{\sum_i^k \sum_{j>i}^k E(S_i, S_j)}, \tag{10}$$

$|S_i|$ は分割 $S_i$ に属するノードの重みの総和であり、$E(S_i, S_j)$ は $S_i$ と $S_j$ の間のエッジの重みの総和である。

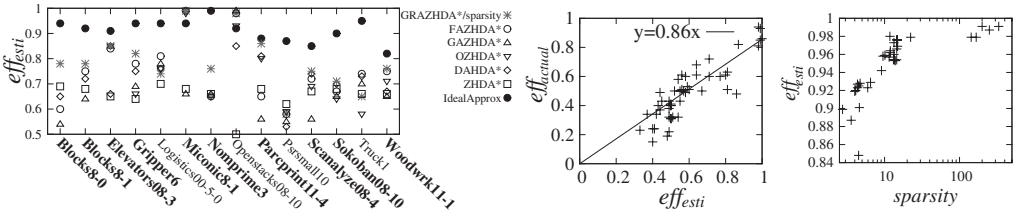
Sparsity は $eff_{esti}$ と強く関係する指標である。

式 9、7 より、sparsity は LB と CO を同時に考慮に入れた指標であると分かる。$\prod_i^k |S_i|$ は LB に対応し、$\sum_i^k \sum_{j>i}^k E(S_i, S_j)$ は CO に対応する。

Sparsity はコンピュータネットワークにおける仕事分配の指標として使われているが [17, 13]、我々の知る限りでは非明示的なグラフに対して用いられた研究はない。

### 5.0.1 実験：Sparsity と $eff_{esti}$ の相関

Sparsity と推定効率 $eff_{esti}$ の相関を見るため、グラフ分割ソフト METIS [14] を用いて図 1a にあるインスタンスの探索空間の分割をした。各インスタンスは 3 回 METIS によって分割を行った（異なるランダムシードを用いてランダムに選択したペアのノードを同じ分割に入れる制約を入れることによって sparsity を低減させた）。図 1c はランダムな制約下の METIS による分割の sparsity と $eff_{esti}$ の比較である。両者には相関が見られる。よって、$sparsity$ を最大化する分割は $eff_{esti}$ も効率化し、実際の実行時間も効率化されると考えられる。




(a) Comparison $eff_{esti}$ for various work distribution methods　(b) $eff_{esti}$ vs. $eff_{actual}$　(c) $sparsity$ vs. $eff_{esti}$

図 1: 図 1a は $c = 1.0, p = 48$ の場合の $eff_{esti}$ の比較である。**太字**は GRAZHDA*が $eff_{esti}$ が最大であるインスタンスを示す (IdealApprox を除く)。図 1b は $c = 1.0, p = 48$ の場合の $eff_{esti}$ と実際の実行時間の比較である。最小二乗法による回帰では $eff_{actual} = 0.86 \cdot eff_{esti}$、残差分散は 0.013 となった。図 1c は sparsity と $eff_{esti}$ の比較である。各インスタンスで 3 回異なる制約下で METIS による分割を行い、sparsity の低減が $eff_{esti}$ にどのような影響を与えるかを評価した。

## 6 Graph Partitioning-Based Abstract Feature Generation (GRAZHDA*)

$eff_{esti}$ が実際の効率をうまく見積もることができ、かつ sparsity と $eff_{esti}$ に強い相関が見られたことから、sparsity を最小化する状態空間の分割は実行速度を近似的に最適化すると言える。

しかしながら状態空間グラフを直接分割するのは現実的ではない。A*探索が扱うグラフは巨大な非明示的なグラフであるためである。明に状態空間グラフを得るためにはまず状態空間グラフを探索しなければならないが、これは探索問題そのものの解も含むことになる。

そのため、状態空間グラフではなくドメイン記述 (e.g. PDDL, SAS+) から抽出できる domain transition graph (DTG) を分割することで最適戦略を近似する *Graph partitioning-based Abstract Zobrist HDA** (**GRAZHDA**) を提案する。

アトムの集合 $x \in X$ に対して domain transition graph (DTG) $\mathcal{D}_x(E, V)$ は $V$ がアトムの集合を表し $E$ が可能な遷移の集合を表す有向グラフである [12]。つまりあるオペレータ $o$ が存在し $v \in del(o)$ かつ $v' \in add(o)$ である場合、$(v, v') \in E$ である。我々の実装では SAS+変数をアトム集合とする DTG を使った。

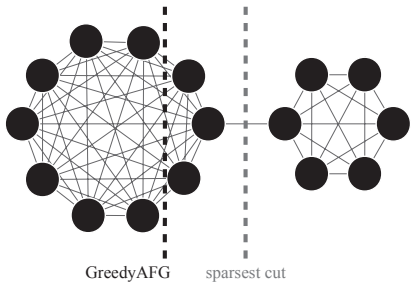



図 2: ロジスティックドメインにおける sparsest cut と GreedyAFG による domain transition graph の分割。

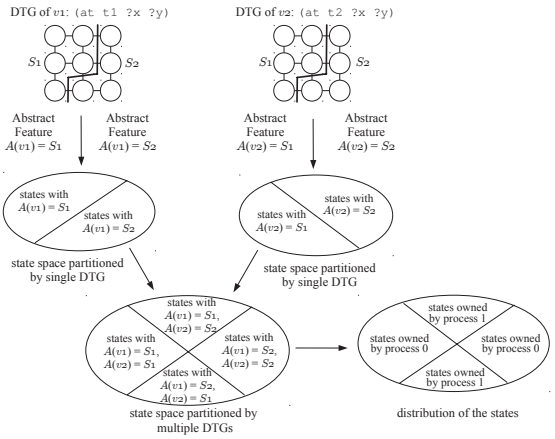



図 3: 分割された DTG とそれらの XOR を取ることによって生成される状態空間全体の分割。

図 2 は `logistics` (ロジスティック) ドメインで荷物の場所を表す DTG を sparsest-cut で分割したものである。

Sparsity を最大化することによって 1 つしかエッジを切らずに良いロードバランスを保つことができている。

GRAZHDA*は DTG の各分割を AZH フレームワークの abstract feature としてとして扱い、各 abstract feature にハッシュ値が与えられる。

AZH による状態のハッシュ値はその状態の abstract feature のハッシュ値の XOR で計算されるため (Eqn 6)、(状態空間における) 2 つのノードが異なる分割に入るのはひとつでも異なる DTG の分割に入っている場合である (Figure 3)。GRAZHDA*は $n$ 個の DTG を元に $2^n$ 個の分割を生成し、それらの分割は $p$ 個のプロセスに割り当てられる (e.g. 分割の ID を $p$ で割った余りを取る)。

DTG の分割が状態空間全体の分割の良い近似とするため、DTG のエッジの重みを

$$e = \frac{\text{\# ground actions which correspond to the transition}}{\text{\# ground actions}} \tag{11}$$

とした。DTG は多くの場合 10 ノード未満なので最適な sparsest-cut は branch-and-bound で簡単に計算することが出来る。

## 7 実験評価

表 1: $eff_{actual}$, $eff_{esti}$, 高速化効率の平均 (spdup), 通信・探索オーバーヘッド (CO, SO) の比較。6 計算ノード、48 プロセッサによる 10 回の実行の平均値。$eff_{esti}$ ($eff_{actual}$) が最も良かった場合は $eff_{esti}$ ($eff_{actual}$) を**太字**にしてある。$eff_{esti}$ が最大の手法と $eff_{actual}$ が最大の手法が同じである場合はインスタンス名を**太字**にしてある。

| Instance | A* | | GRAZHDA*/sparsity | | | | | FAZHDA* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | time | expd | $eff_{actual}$ | $eff_{esti}$ | spdup | CO | SO | $eff_{actual}$ | $eff_{esti}$ | spdup | CO | SO |
| **Blocks10-0** | 129.29 | 11065451 | **0.57** | **0.57** | **27.17** | 0.28 | 0.38 | 0.54 | 0.43 | 26.02 | 0.70 | 0.35 |
| **Blocks11-1** | 813.86 | 52736900 | **0.71** | **0.53** | **34.25** | 0.66 | 0.15 | 0.71 | 0.50 | 34.25 | 0.66 | 0.15 |
| Elevators08-5 | 165.22 | 7620122 | 0.34 | 0.51 | 16.43 | 0.47 | 0.33 | 0.26 | 0.49 | 12.34 | 0.32 | 0.51 |
| **Elevators08-6** | 453.21 | 18632725 | 0.45 | 0.50 | 21.47 | 0.49 | 0.37 | 0.38 | 0.36 | 18.05 | 0.52 | 0.81 |
| **Gripper8** | 517.41 | 50068801 | 0.56 | 0.60 | 26.67 | 0.50 | 0.15 | **0.57** | **0.63** | **27.45** | 0.43 | 0.10 |
| **Logistics00-10-1** | 559.45 | 38720710 | **0.94** | **0.70** | **45.16** | 0.43 | 0.01 | 0.91 | 0.61 | 43.85 | 0.57 | 0.02 |
| Miconic11-0 | 232.07 | 12704945 | 0.87 | 0.95 | 41.97 | 0.01 | 0.07 | 0.88 | 0.91 | 42.43 | 0.01 | 0.06 |
| **Miconic11-2** | 262.01 | 14188388 | **0.94** | **0.97** | **45.26** | 0.01 | 0.05 | 0.93 | 0.92 | 44.87 | 0.01 | 0.05 |
| **NoMprime5** | 309.14 | 4160871 | **0.50** | **0.58** | **23.95** | 0.80 | -0.04 | 0.48 | 0.53 | 22.87 | 0.79 | -0.05 |
| NoMystery10 | 179.52 | 1372207 | **0.72** | 0.61 | **34.80** | 0.51 | 0.12 | 0.48 | **0.75** | 22.99 | 0.24 | -0.44 |
| Openstacks08-19 | 282.45 | 15116713 | 0.51 | 0.59 | 24.67 | 0.27 | 0.34 | 0.42 | 0.58 | 20.00 | 0.24 | 0.37 |
| Openstacks08-21 | 554.63 | 19901601 | 0.53 | 0.65 | 25.23 | 0.17 | 0.35 | 0.52 | 0.62 | 24.97 | 0.15 | 0.35 |
| **Parcprinter11-11** | 307.19 | 6587422 | **0.42** | **0.54** | **20.26** | 0.26 | 0.55 | 0.27 | 0.49 | 13.08 | 0.26 | 0.61 |
| Parking11-5 | 237.05 | 2940453 | **0.62** | 0.55 | **29.75** | 0.40 | 0.34 | 0.62 | 0.54 | 29.67 | 0.63 | 0.11 |
| Pegsol11-18 | 801.37 | 106473019 | 0.44 | **0.72** | 21.03 | 0.39 | 0.02 | 0.44 | 0.71 | 20.97 | 0.39 | 0.00 |
| **PipesNoTk10** | 157.31 | 2991859 | **0.33** | **0.52** | **15.73** | 0.98 | 0.01 | 0.33 | 0.49 | 15.64 | 0.98 | 0.01 |
| PipesTk12 | 321.55 | 15990349 | 0.70 | **0.66** | 33.78 | 0.46 | 0.05 | **0.83** | 0.65 | **39.65** | 0.46 | 0.03 |
| PipesTk17 | 356.14 | 18046744 | 0.92 | **0.65** | 43.92 | 0.54 | 0.01 | **0.94** | 0.63 | **45.03** | 0.54 | 0.01 |
| Rovers6 | 1042.69 | 36787877 | **0.86** | **0.79** | **41.17** | 0.15 | 0.14 | 0.84 | 0.72 | 40.48 | 0.15 | 0.17 |
| **Scanalyzer08-6** | 195.49 | 10202667 | **0.69** | **0.92** | **32.92** | 0.12 | 0.01 | 0.63 | 0.86 | 30.31 | 0.12 | 0.01 |
| **Scanalyzer11-6** | 152.92 | 6404098 | **0.91** | **0.78** | **43.83** | 0.16 | 0.13 | 0.57 | 0.63 | 27.31 | 0.18 | 0.34 |
| **Average** | 382.38 | 21557805 | **0.64** | **0.62** | **30.92** | 0.38 | 0.17 | 0.60 | 0.61 | 28.68 | 0.40 | 0.17 |
| Total walltime | 8029.97 | 452713922 | | | | **277.91** | | | | | | 301.38 |

| | GAZHDA* | | | | | OZHDA* | | | | | DAHDA* | | | | | ZHDA* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $eff_{actual}$ | $eff_{est}$ | spdup | CO | SO | $eff_{actual}$ | $eff_{est}$ | spdup | CO | SO | $eff_{actual}$ | $eff_{est}$ | spdup | CO | SO | $eff_{actual}$ | $eff_{est}$ | spdup | CO | SO |
| **Blocks10-0** | 0.45 | 0.44 | 21.81 | 0.99 | 0.12 | 0.32 | 0.37 | 15.47 | 0.98 | 0.34 | 0.52 | 0.47 | 25.11 | 0.88 | 0.08 | 0.31 | 0.48 | 14.93 | 0.98 | 0.30 |
| **Blocks11-1** | 0.61 | 0.48 | 29.20 | 0.99 | 0.03 | 0.61 | 0.47 | 29.20 | 0.99 | 0.03 | 0.52 | 0.43 | 24.88 | 0.91 | 0.21 | 0.58 | 0.48 | 27.98 | 0.98 | 0.07 |
| Elevators08-5 | **0.61** | 0.58 | **29.35** | 0.65 | -0.00 | 0.46 | **0.64** | 21.86 | 0.09 | 0.44 | 0.57 | 0.51 | 27.59 | 0.83 | -0.03 | 0.57 | 0.47 | 27.54 | 0.98 | -0.03 |
| **Elevators08-6** | **0.72** | **0.76** | **34.52** | 0.24 | -0.09 | 0.68 | 0.56 | 32.70 | 0.41 | 0.22 | 0.32 | 0.39 | 15.28 | 0.88 | 0.31 | 0.38 | 0.49 | 18.19 | 0.96 | 0.06 |
| **Gripper8** | 0.46 | 0.50 | 21.86 | 0.81 | 0.06 | 0.52 | 0.44 | 24.77 | 0.98 | 0.14 | 0.45 | 0.45 | 21.80 | 0.98 | 0.08 | 0.45 | 0.47 | 21.66 | 0.98 | 0.08 |
| **Logistics00-10-1** | 0.24 | 0.42 | 11.68 | 0.85 | 0.25 | 0.24 | 0.43 | 11.68 | 0.85 | 0.25 | 0.36 | 0.53 | 17.52 | 0.84 | 0.00 | 0.34 | 0.48 | 16.09 | 0.99 | 0.00 |
| Miconic11-0 | 0.27 | 0.53 | 13.15 | 0.53 | 0.24 | 0.79 | **0.96** | 37.86 | 0.02 | 0.02 | **0.96** | 0.91 | **46.05** | 0.01 | 0.08 | 0.15 | 0.48 | 7.40 | 0.96 | 0.13 |
| **Miconic11-2** | 0.18 | 0.37 | 8.53 | 0.53 | 0.74 | 0.77 | 0.90 | 36.86 | 0.02 | 0.07 | 0.70 | 0.81 | 33.81 | 0.01 | 0.18 | 0.31 | 0.48 | 14.67 | 0.96 | 0.05 |
| **NoMprime5** | 0.39 | 0.48 | 18.55 | 0.95 | -0.06 | 0.35 | 0.51 | 16.66 | 0.94 | 0.00 | 0.38 | 0.49 | 18.46 | 0.90 | -0.05 | 0.35 | 0.47 | 16.63 | 0.98 | -0.02 |
| NoMystery10 | 0.40 | 0.66 | 18.98 | 0.42 | -0.07 | 0.45 | 0.50 | 21.61 | 0.74 | 0.11 | 0.59 | 0.60 | 28.41 | 0.60 | -0.07 | 0.45 | 0.49 | 21.68 | 0.99 | -0.07 |
| Openstacks08-19 | 0.46 | 0.58 | 22.14 | 0.38 | 0.21 | 0.36 | 0.55 | 17.11 | 0.34 | 0.32 | 0.51 | **0.66** | 24.54 | 0.24 | 0.18 | **0.54** | 0.47 | **25.99** | 0.99 | -0.05 |
| Openstacks08-21 | 0.53 | 0.65 | 25.67 | 0.15 | 0.31 | **0.82** | 0.49 | **39.34** | 0.92 | 0.05 | 0.56 | **0.68** | 26.72 | 0.13 | 0.28 | 0.81 | 0.51 | 39.06 | 0.92 | -0.00 |
| **Parcprinter11-11** | 0.35 | 0.40 | 16.85 | 0.74 | 0.41 | 0.33 | 0.34 | 15.98 | 0.82 | 0.56 | 0.15 | 0.15 | 7.00 | 0.19 | 4.38 | 0.40 | 0.48 | 19.15 | 0.97 | 0.08 |
| Parking11-5 | 0.59 | 0.49 | 28.43 | 0.98 | 0.02 | 0.56 | 0.46 | 26.76 | 0.97 | 0.07 | 0.60 | **0.59** | 28.84 | 0.52 | 0.07 | 0.56 | 0.47 | 27.09 | 0.98 | 0.04 |
| Pegsol11-18 | 0.34 | 0.53 | 16.22 | 0.77 | 0.05 | **0.55** | 0.71 | **26.17** | 0.34 | -0.03 | 0.46 | 0.70 | 22.16 | 0.34 | -0.01 | 0.35 | 0.47 | 16.97 | 0.98 | 0.03 |
| **PipesNoTk10** | 0.32 | 0.50 | 15.58 | 0.98 | 0.01 | 0.32 | 0.48 | 15.22 | 0.98 | 0.02 | 0.32 | 0.48 | 15.58 | 0.98 | 0.01 | 0.07 | 0.48 | 3.22 | 0.98 | -0.44 |
| PipesTk12 | 0.41 | 0.48 | 19.84 | 0.99 | 0.01 | 0.45 | 0.49 | 21.40 | 0.88 | 0.04 | 0.52 | 0.57 | 25.12 | 0.67 | 0.00 | 0.41 | 0.48 | 19.78 | 0.98 | 0.00 |
| PipesTk17 | 0.56 | 0.50 | 26.64 | 0.98 | 0.00 | 0.60 | 0.52 | 28.82 | 0.88 | 0.00 | 0.65 | 0.60 | 31.16 | 0.60 | 0.01 | 0.55 | 0.49 | 26.27 | 0.98 | 0.00 |
| Rovers6 | 0.70 | 0.61 | 33.49 | 0.56 | 0.01 | 0.85 | 0.71 | 41.00 | 0.31 | 0.03 | 0.53 | 0.73 | 25.48 | 0.05 | 0.26 | 0.63 | 0.53 | 30.01 | 0.76 | 0.00 |
| **Scanalyzer08-6** | 0.42 | 0.54 | 20.28 | 0.77 | 0.01 | 0.49 | 0.58 | 23.70 | 0.66 | 0.01 | 0.44 | 0.51 | 21.23 | 0.94 | 0.00 | 0.34 | 0.48 | 16.54 | 0.98 | 0.01 |
| **Scanalyzer11-6** | 0.34 | 0.41 | 16.36 | 0.65 | 0.49 | 0.81 | 0.68 | 38.82 | 0.30 | 0.09 | 0.41 | 0.44 | 19.51 | 0.50 | 0.46 | 0.42 | 0.48 | 20.36 | 0.98 | 0.05 |
| **Average** | 0.45 | 0.51 | 21.39 | 0.71 | 0.13 | 0.54 | 0.53 | 25.86 | 0.64 | 0.13 | 0.50 | 0.47 | 24.11 | 0.57 | 0.31 | 0.43 | 0.49 | 20.53 | 0.96 | 0.01 |
| Total walltime | | | | | 398.75 | | | | | 331.18 | | | | | 377.86 | | | | | 433.23 |

図 1a は $eff_{esti}$ の比較である (実験条件は 4.2.1 節)。*IdealApprox* は状態空間グラフのノードとエッジを列挙し直接 METIS [14] によって分割した場合の $eff_{esti}$ の値である。これは状態空間グラフを列挙する必要があるため現実的な手法ではないが、ほぼ最適なグラフ分割による $eff_{esti}$ の値と提案手法のそれを比較するためである。

IdealApprox が一番 $eff_{esti}$ が大きかった。現実的な手法の中では GRAZHDA*/sparsity が全体として最も $eff_{esti}$ が良かった。4.2.1 節で見たように、$eff_{esti}$ は実際の高速化効率の良い推定値にな

表 2: $eff_{actual}$, $eff_{esti}$, 高速化効率の平均 (spdup), 通信・探索オーバーヘッド (CO, SO) の比較。128 バーチャルプロセッサ (32 個の m1.xlarge EC2 instances) による 10 回の実行の平均値。$eff_{esti}$ ($eff_{actual}$) が最も良かった場合は $eff_{esti}$ ($eff_{actual}$) を**太字**にしてある。$eff_{esti}$ が最大の手法と $eff_{actual}$ が最大の手法が同じである場合はインスタンス名を**太字**にしてある。

| Instance | A* | GRAZHDA*/sparsity | | | FAZHDA* | | |
|---|---|---|---|---|---|---|---|
| | expd | time | CO | SO | time | CO | SO |
| Airport18 | 48782782 | 102.34 | 0.59 | 0.49 | **95.48** | 0.59 | 0.29 |
| Blocks11-0 | 28664755 | **12.40** | 0.42 | 0.37 | 22.86 | 0.68 | 0.53 |
| Blocks11-1 | 45713730 | **17.21** | 0.42 | 0.25 | 32.60 | 0.66 | 0.82 |
| Elevators08-7 | 74610558 | **51.90** | 0.54 | 0.25 | 121.90 | 0.55 | 0.26 |
| Gripper9 | 243268770 | **78.90** | 0.42 | 0.01 | 82.90 | 0.43 | 0.06 |
| Openstacks08-21 | 19901601 | 6.30 | 0.23 | 0.06 | 5.76 | 0.19 | -0.05 |
| Openstacks11-18 | 115632865 | **33.10** | 0.24 | -0.14 | 33.25 | 0.23 | -0.12 |
| Pegsol08-29 | 287232276 | 58.85 | 0.44 | 0.16 | 81.75 | 0.42 | 0.55 |
| PipesNoTk16 | 60116156 | 120.64 | 0.94 | 0.84 | **106.28** | 0.94 | 0.72 |
| Trucks6 | 19109329 | **8.01** | 0.17 | 0.46 | 51.51 | 0.19 | 0.34 |
| Average | 99361115 | **43.03** | 0.42 | 0.25 | 59.87 | 0.48 | 0.39 |
| Total walltime | 894250040 | | | **387.31** | | | 538.81 |

| Instance | GAZHDA* | | | OZHDA* | | | DAHDA* | | | ZHDA* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | time | CO | SO | time | CO | SO | time | CO | SO | time | CO | SO |
| Airport18 | 128.22 | 0.98 | 0.02 | 123.09 | 0.90 | 0.56 | 143.27 | 0.92 | 0.36 | 106.80 | 0.99 | 0.02 |
| Blocks11-0 | 21.75 | 0.98 | 0.65 | 21.70 | 0.99 | 0.70 | 20.29 | 0.95 | 0.88 | 29.19 | 0.99 | 0.35 |
| Blocks11-1 | 25.84 | 0.98 | 0.56 | 24.84 | 0.86 | 0.78 | 29.52 | 0.94 | 0.83 | 36.04 | 1.00 | 0.52 |
| Elevators08-7 | 61.16 | 0.70 | 0.05 | 86.65 | 0.07 | 0.22 | 52.09 | 0.96 | 0.18 | 59.88 | 1.00 | 0.04 |
| Gripper9 | 85.98 | 1.00 | 0.16 | 90.98 | 0.98 | 0.20 | 95.72 | 1.00 | 0.15 | 105.78 | 1.00 | 0.17 |
| Openstacks08-21 | **5.67** | 0.71 | -0.35 | 40.06 | 0.96 | 0.00 | 6.94 | 0.69 | -0.17 | 14.65 | 1.00 | -0.09 |
| Openstacks11-18 | 71.34 | 0.77 | -0.09 | 79.34 | 0.81 | -0.00 | 84.67 | 0.76 | 0.01 | 49.97 | 1.00 | -0.53 |
| Pegsol08-29 | 98.53 | 0.98 | 0.06 | **54.13** | 0.34 | 0.13 | 108.17 | 1.00 | 0.11 | 120.27 | 0.98 | 0.16 |
| PipesNoTk16 | 108.28 | 0.95 | 0.78 | 120.21 | 0.99 | 0.73 | 125.37 | 1.00 | 0.72 | 149.96 | 1.00 | 0.73 |
| Trucks6 | 30.22 | 0.94 | 0.41 | 32.22 | 0.96 | 0.57 | 17.19 | 0.53 | 0.43 | 28.22 | 1.00 | 0.34 |
| Average | 56.53 | 0.89 | 0.29 | 61.13 | 0.77 | 0.41 | 60.00 | 0.87 | 0.36 | 66.00 | 1.00 | 0.29 |
| Total walltime | | | 508.77 | | | 550.13 | | | 539.96 | | | 593.96 |

るので、GRAZHDA*/sparsity は他の手法よりも高速であることが示唆される。実際、表 1 によると GRAZHDA*/sparsity が CO と SO の良いトレードオフを保ち、既存手法と比較して最も良い平均の高速化効率を達成した。

**クラウドクラスタにおける実験結果:** 48 コアのクラスタに加え、Amazon EC2 のクラウドクラスタにおける実験も行った。クラスタは 128 個のバーチャルコア (vCPUs)、合計 480GB のメモリからなる (32 個の m1.xlarge EC2 インスタンス。各 4 vCPUs、3.75 GB RAM/core)。

このクラウドクラスタは並列探索のための環境としてはあまり適さない安価な環境であり、ネットワークの効率が不安定である [8]。様々な実行環境における仕事分配手法のパフォーマンスを比較するためにこの環境を選んだ。表 2 は実験結果である。48 コアクラスタと同様、GRAZHDA*/sparsity が最も良い高速化効率を達成した。

## 8 結論

本研究では HDA*のための新しい仕事分配手法のフレームワークを提案した。主な貢献は以下の 3 点である：1. 実際の実行時間を推定することが出来る、通信・探索オーバーヘッドに基づく評価指標 $eff_{esti}$ の提案 2. $eff_{esti}$ の最大化につながる sparsest cut による定式化 3. 最適な仕事分配手法を近似する GRAZHDA*の提案。GRAZHDA*が実験的に高速であることを示した。

先行研究と比較して大幅な高速化が実現されたが、提案手法にはいくつか改善の余地がある。本研究ではプロセス間の通信オーバーヘッドは一定であると仮定している。しかしながら大規模並列環境ではこの仮定はまず成立しない。ハードウェアのローカリティを考慮した仕事分配手法によってより高速な並列 A*が実装できると考えられる。

## 参考文献


[1] Aydin Buluc, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. *arXiv preprint arXiv:1311.3144*, 2015.

[2] Ethan Burns, Sofia Lemons, Wheeler Ruml, and Rong Zhou. Best-first heuristic search for multi-core machines. *Journal of Artificial Intelligence Research (JAIR)*, Vol. 39, pp. 689–743, 2010.

[3] Alex Fukunaga, Adi Botea, Yuu Jinnai, and Akihiro Kishimoto. A survey of parallel A*. *arXiv preprint arXiv:1708.05296*, 2017.

[4] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, Vol. 4, No. 2, pp. 100–107, 1968.

[5] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, Vol. 26, pp. 191–246, 2006.

[6] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM (JACM)*, Vol. 61, No. 3, p. 16, 2014.

[7] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel computing*, Vol. 26, No. 12, pp. 1519–1534, 2000.

[8] Alexandru Iosup, Simon Ostermann, M Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick HJ Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, No. 6, pp. 931–945, 2011.

[9] Yuu Jinnai and Alex Fukunaga. Abstract Zobrist hashing: An efficient work distribution method for parallel best-first search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 717–723, 2016.

[10] Yuu Jinnai and Alex Fukunaga. Automated creation of efficient work distribution functions for parallel best-first search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2016.

[11] Yuu Jinnai and Alex Fukunaga. On work distribution functions for parallel best-first search. *Journal of Artificial Intelligence Research*, 2017.

[12] Peter Jonsson and Christer Bäckström. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, Vol. 100, No. 1, pp. 125–176, 1998.

[13] Sangeetha Abdu Jyothi, Ankit Singla, P Godfrey, and Alexandra Kolla. Measuring and understanding throughput of network topologies. *arXiv preprint arXiv:1402.2531*, 2014.

[14] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, Vol. 20, No. 1, pp. 359–392, 1998.

[15] Akihiro Kishimoto, Alex S. Fukunaga, and Adi Botea. Scalable, parallel best-first search for optimal sequential planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 201–208, 2009.

[16] 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.

[17] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)*, Vol. 46, No. 6, pp. 787–832, 1999.

[18] Rong Zhou and Eric A Hansen. Parallel structured duplicate detection. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1217–1223, 2007.

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

[20] A. L. Zobrist. A new hashing method with applications for game playing. Technical report, Dept of CS, Univ. of Wisconsin, Madison, 1970. Reprinted in *International Computer Chess Association Journal*, 13(2):169-173, 1990.