日本オペレーションズ・リサーチ学会和文論文誌
Transactions of the Operations Research Society of Japan
Vol. 54, 2011, pp. 58–83
© 日本オペレーションズ・リサーチ学会

大規模最短路問題に対するダイクストラ法の高速化

安井 雄一郎 中央大学
藤澤 克樹 中央大学
笹島 啓史 中央大学
後藤 和茂 マイクロソフト

(受理 2010 年 3 月 31 日; 再受理 2011 年 2 月 24 日)

和文概要　最短路問題はネットワーク上の経路探索などの多くの応用を持ち，また他の最適化問題の子問題として用いられることも多く，適用範囲の広い組合せ最適化問題である．そのため最短路問題を高速に解くことの重要性は非常に大きくなってきている．最短路問題に対する解法としてはダイクストラ法などの安定的かつ効率的な高速アルゴリズムが存在するが，実問題は非常に大規模になるためさらなる高速化が不可欠である．そこで本論文では大規模最短路問題に対し，計算機のメモリ階層構造を考慮しつつ汎用的かつ効率的に高速化を行うための実装方法を示す．さらに論文中では計算機のメモリ階層構造における律速箇所の特定を行うための汎用的な解析方法を示し，高速化の有用性を検証していく．本手法により実装されたバイナリ・ヒープを適用したダイクストラ法は，実行性能，安定性，メモリ要求量などを他の実装と比較すると総合的に最も優れているといえる．また本実装を用いた大規模最短路問題に対するオンライン・ソルバーについても説明を行う．

キーワード: アルゴリズム，グラフ理論，計算機，最適化，情報技術，ネットワークフロー

1. はじめに

最短路問題は最も基本的かつ重要な組合せ最適化問題のひとつであり，2006 年に 9th DIMACS Implementation Challenge - Shortest Paths (9th DIMACS) [1, 8, 9] が開催されるなど，現在も盛んに研究が行われている．9th DIMACS で発表された先行研究 [3, 7, 15] は，全米道路ネットワーク* 上の交差点を点，交差点間の道路を枝とした非常に大規模なグラフ (約 2400 万点，約 5800 万枝) を対象とし，枝長が頻繁に変化しないという特性を利用して数十分から数時間の前処理を行い，経路探索要求 (クエリ) に高速に対応するように設計されている．しかしながら，渋滞情報・事故情報などを考慮した経路探索，リアルタイム性が非常に重要となる大規模災害時の避難経路探索，他の最適化問題の子問題として実行される経路探索など，前処理を行わずに高速な探索を要求される機会は少なくない．

非負の枝長を持つグラフに対する効率的なアルゴリズムとしてダイクストラ法 [11] が有名であるが，アルゴリズム中の "次探索点の選択" がボトルネックとなることが知られている．そこで，探索候補点集合に優先キューと呼ばれるデータ構造を適用することで改善が行われてきた [4, 5, 10, 12, 14, 16, 20]．中でもマルチレベル・バケット [14] は，理論的にも実験的にも高速な優先キューであることが知られている．しかしながら，メモリ要求量が大きく，本論文でも検証するようにマルチコア・プロセッサ上において性能を十分に引き出すように設計されていない．そこで本研究では，実行性能，グラフ特性に対する安定性，メモリ要求量，並列実行など，総合的に最も優れたソルバーを開発することを目的とし，バイナリ・ヒープを適用したダイクストラ法に対して計算機のメモリ階層構造を考慮した高速化 [17–19] を行った．論文中でも示すように本実装方法は，特定の計算機環境に依存するこ

*http://www.census.gov/geo/www/tiger/

となく汎用的に適用可能である．さらにメモリ階層構造上の律速箇所の特定を行うための解析方法を提案し，本実装の性能を検証していく．

2. ダイクストラ法に対する優先キューの適用

2.1. 最短路問題に対するダイクストラ法

各枝に非負の重みを持つ有向グラフ $G = (V, E)$ に対し次の 2 種類の最短路問題を考える:

- 1 対全最短路問題 (Single-Source Shortest Path problem; SSSP):
  - 入力: 始点 $s \in V$,
  - 出力: 始点から各点までの最短距離と最短路.
- 1 対 1 最短路問題 (Point-to-Point Shortest Path problem; P2PSP):
  - 入力: 始点 $s \in V$，終点 $t \in V$,
  - 出力: 始点から終点までの最短距離と最短路.

ダイクストラ法では各点 $v \in V$ に対し，次のような作業領域が必要となる:

- $d(v)$: 始点からの距離,
- $\pi(v)$: 最短路における直前の訪問点 (確定されていない場合では `nil` とする),
- $S(v)$: 探索状況を示すフラグ (`unreached`: 未探索，`labeled`: 探索済，`scanned`: 確定).

点 s を始点とする最短路問題を計算する際には，$d(s) = 0$，$\pi(s) = \mathtt{nil}$，$S(s) = \mathtt{scanned}$，$d(v) = \infty$，$\pi(v) = \mathtt{nil}$，$S(v) = \mathtt{unreached}$ $(\forall v \in V \setminus \{s\})$ と初期化し探索を開始する．各反復時，探索点 v の距離 $d(v)$ と接続されている枝 $(v, w) \in E$ の長さ (重み) $l(v, w)$ に対して，$d(v) + l(v, w) < d(w)$ が成立すれば $d(w) \leftarrow l(v, w) + d(v)$ と距離の更新を行い，$S(w)$ が `unrearched` であれば `labeled` とする．探索点に接続されている枝をすべて探索し終わると $S(v) = \mathtt{scanned}$ とし，フラグが `labeled` かつ距離が最小の点を新たに探索点として探索を繰り返す．1 対 1 最短路問題では終点 t のフラグ $S(t)$ が `scanned` に，1 対全最短路問題では全点のフラグ $S(v)$ $(\forall v \in V)$ が `scanned` になることが終了条件である．各点は高々 1 回だけ探索点となり，各枝も高々1 回だけ探索される．ダイクストラ法の効率は次探索点決定の方法に依存するため，探索点候補 (フラグが `labeled` である点集合) に対し適用した優先キューの特性にアルゴリズムが依存する．

2.2. ダイクストラ法に対する優先キュー

最短路問題に対する優先キュー Q は，`insert`，`decrease-key` (もしくは `delete`，`insert`), `extract-min` という操作に対応したデータ構造であり，大きく分類すると “ヒープを用いた優先キュー” と “バケットを用いた優先キュー” の 2 種類になる．各操作の詳細は次のとおりである:

- `insert`: 点 $v \in V$ を，距離 $d(v)$ を優先度として優先キュー Q に挿入する,
- `delete`: 点 $v \in Q$ を優先キューから削除する,
- `decrease-key`: 点 $v \in Q$ に対して，距離 $d(v)$ を $d'(v)$ $(d'(v) < d(v))$ に更新する,
- `extract-min`: 距離 $d(v)$ が最小の点 $v \in Q$ を取り出す.

2.2.1. ヒープを用いた優先キュー

ヒープを用いた優先キューでは，優先度の大小関係により木構造を構成するため，扱うデータ値の範囲によって性能が左右されることは少なく，実行時間やメモリ要求量が安定的である．最も基本的なバイナリ・ヒープ (binary heap または 2-ary heap)[20] においては，各親は 2 つ

©日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

の子を持ち，親子間には「親の優先度 > 子の優先度」が常に成り立つ (図 1 参照)．また根に配置されているデータが最も優先度が高いように構成されている．insert，decrease-key，extract-min 操作の計算量はそれぞれ $O(\log_2 n)$ となり，同数のスワップ操作 (木構造を構成するためデータの入れ替え) が必要になる．

2.2.2. バケットを用いた優先キュー

バケットを用いた優先キューでは，優先度の値に対して予め決定した規則に従いデータを配置させるため，扱う値の範囲に実行時間やメモリ要求量が依存してしまう．最も基本的な 1 レベル・バケット (1-level buckets または Dial's algorithm)[10] は，探索候補点集合の距離ラベルの取りうる値の範囲の幅が $C+1$ (C: 最大枝長) となることを利用して，$C+1$ 個のバケットから "$d(v) \mod (C+1)$" の値によって 1 つのバケットを決定する (図 2 参照)．このとき同一のバケットに格納される点は距離ラベルが等しい．計算量が $O(1)$ である insert や decrease-key に対し，extract-min は $O(C+1)$ であるため，最大枝長 C が大きくなるに従い必要なバケットの数も増加し性能が低下する．また，9th DIMACS ベンチマーク・ソルバー (mbp)[8] で使用されたマルチレベル・バケット (multi-level buckets)[14] は，2 の累乗で分類するため優先度の値に性能が依存しにくく，実行時間やメモリ要求量が安定的である．

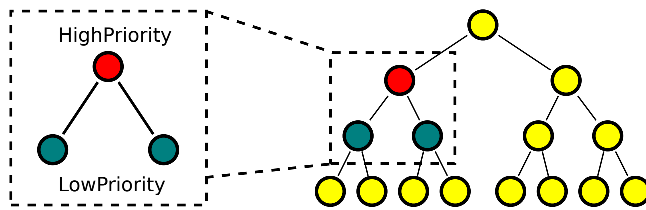


図 1: バイナリ・ヒープの例

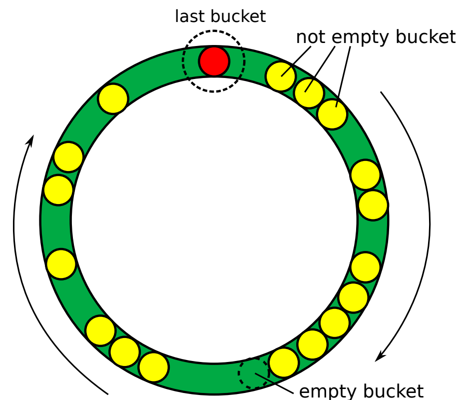


図 2: 1 レベル・バケットの例

2.3. 優先キューの特性

次にヒープを用いた優先キューとバケットを用いた優先キューの典型的な特性を示す．DIKH (4-ary heap)，DIKB (1-level buckets)[4]，mbp[14] に対し，9th DIMACS ベンチマーク問題 [9] を用いて比較を行う．実験に用いた長方形 (1 辺を 16 に固定した) 型グラフ Long-C Family (1,048,576 点，4,063,200 枝)，ランダムグラフ Random4-C Family (1,048,576 点，4,194,304 枝)，正方形型グラフ Square-C Family (1,048,576 点，4,190,208 枝) は，それぞれ点数・枝数・枝の接続情報を固定し，図中の横軸パラメータの $i \in \{0, ..., 15\}$ により最大枝長 4^i を決定している (表 1 参照)．各グラフに対し，1 対 1 最短路問題 1 クエリあたりの実行時間を図 3，5，7 に，その際に要求されるメモリ要求量を図 4，6，8 にまとめた．いずれのグラフにおいても DIKB は実行時間やメモリ要求量が最大枝長に依存することが確認できる．一方，DIKH と mbp は最大枝長に対する依存度が低く安定的である．また，本論文による高速化後のダイクストラ法に関する実験結果は Appendix B，C，D，E，F，G に掲載した．なお，計算機環境は Harpertown (表 2 参照) である．

©日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

表 1: 9th DIMACS ベンチマーク問題のグラフ特性

Family	点数	枝数	枝長	次数	詳細
Long-n	2^i	$4 \times 2^i - 32 - 2 \times (2^i/16)$	$[0, 2^i]$	[2, 4]	1 辺を 16 とした長方形型グラフ
Long-C	2^{20}	$4 \times 2^{20} - 32 - 2 \times (2^{20}/16)$	$[0, 4^i]$	[2, 4]	1 辺を 16 とした長方形型グラフ
Random4-n	2^i	4×2^i	$[0, 2^i]$	[1,17]	ランダム型グラフ
Random4-C	2^{20}	4×2^{20}	$[0, 4^i]$	[1,17]	ランダム型グラフ
Square-n	2^i	$4 \times 2^i - 4 \times 2^{i/2}$	$[0, 2^i]$	[2, 4]	正方形型グラフ
Square-C	2^{20}	$4 \times 2^{20} - 4 \times 2^{20/2}$	$[0, 4^i]$	[2, 4]	正方形型グラフ

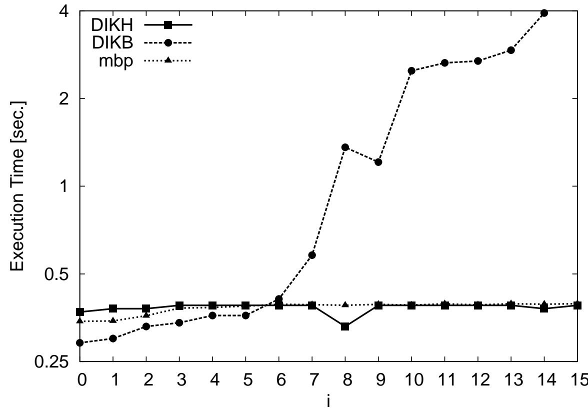


図 3: Long-C.**i**.0.gr: 実行時間 [sec.]

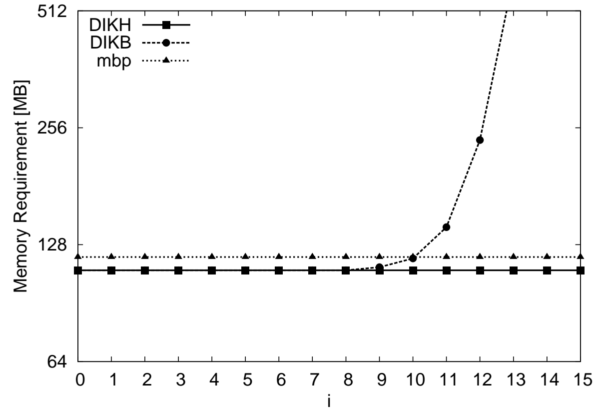


図 4: Long-C.**i**.0.gr: メモリ要求量 [MB]

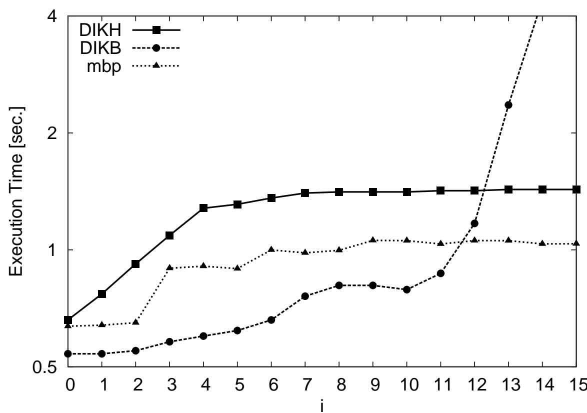


図 5: Random4-C.**i**.0.gr: 実行時間 [sec.]

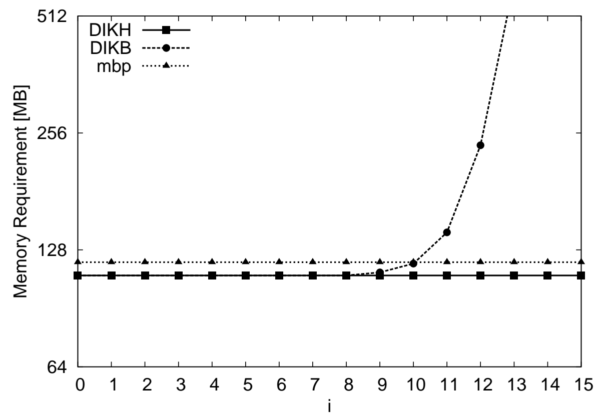


図 6: Random4-C.**i**.0.gr: メモリ要求量 [MB]

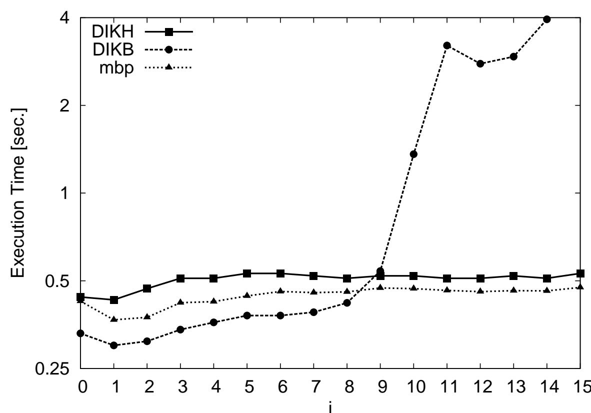


図 7: Square-C.**i**.0.gr: 実行時間 [sec.]

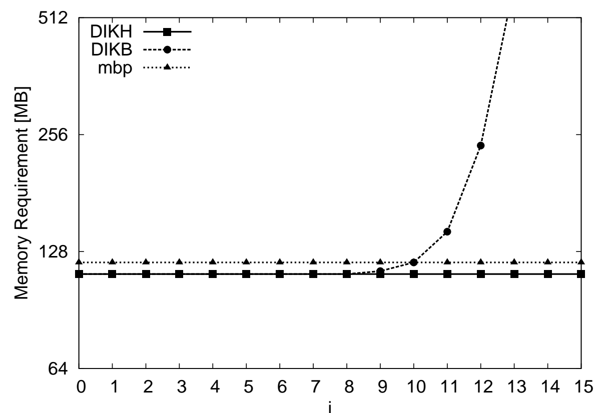


図 8: Square-C.**i**.0.gr: メモリ要求量 [MB]

©日本オペレーションズ・リサーチ学会. 無断複写・複製・転載を禁ず.

表 2: 計算機環境

	プロセッサ	搭載メモリ	OS(Linux)	GCC
Clovertown	Intel Xeon(R) E5345 2.33GHz ×2 (4 コア ×2)	16GB	CentOS 5.4	4.1.2
Harpertown	Intel Xeon(R) X5460 3.16GHz ×2 (4 コア ×2)	48GB	CentOS 5.4	4.1.2
Nehalem-EP	Intel Xeon(R) X5550 2.67GHz ×2 (4 コア ×2)	72GB	Fedora 12	4.4.3
Barcelona	AMD Opteron(tm) 2356 2.30GHz ×2 (4 コア ×2)	36GB	CentOS 5.4	4.1.2
Shanghai	AMD Opteron(tm) 2384 2.70GHz ×2 (4 コア ×2)	36GB	Fedora 12	4.4.3

3. メモリ階層構造を考慮した高速化

3.1. 高速化の方針

本論文ではメモリ階層構造を考慮することにより，特定のアーキテクチャや問題特性に特化させることなく汎用的に高速化を行うことを目的とする．特に最新のマルチコア・プロセッサを搭載した計算機環境を想定し，より多くのプロセッサコア上で効率良く実行するためにメモリ要求量やキャッシュメモリなどの資源の競合を抑えるように設計する．また自由度の高い実装を行うために C 言語を用いる．

性能効率を上げやすい条件としては，“データ移動量に対し演算量がある程度大きいこと (つまり $\frac{\text{演算量}}{\text{データ移動量}}$ の比がある程度大きいこと)” や，“データアクセスが連続的で，中長期的に予測が可能であること” が挙げられるが，大規模最短路問題におけるダイクストラ法は，“データ移動量に対し演算量が非常に小さく” かつ “不連続な領域に対しデータアクセスが広域に及ぶため中長期的な予測が困難” という特性を持っているので性能効率を上げるのは容易ではない．ここで演算量とは，レジスタ上の演算器で行うデータの移動命令以外のすべての演算命令の回数とする．

3.2. メモリ階層構造

本論文では計算機内部が図 9 のようなメモリ階層構造になっていると仮定する [13]．メモリ階層構造では，上位階層であるほどアクセス速度が高速で小容量な記憶領域を，下位階層であるほどアクセス速度が低速で大容量な記憶領域を保持している．特に CPU 内部では，レジスタやキャッシュメモリ，TLB† など非常に高速な記憶領域が存在する．演算処理はレジスタ上でのみ行うことができるが，アクセス速度が非常に高速 (250 pico seconds) な反面，容量が非常に小さい．また主記憶装置 (RAM) は数 Gbytes 以上と非常に大容量であるがアクセス速度はレジスタと比較すると極めて低速である (100 nano seconds)．そのため最適化分野に限らずソフトウェアの高速化のためには，演算量とデータ移動量の割合を考慮してデータを適切に配置し，レジスタと主記憶装置の中間に位置するキャッシュメモリを有効に利用することは非常に重要である．L2 (あるいは L3) キャッシュメモリは，比較的高速で数 Mbytes という大きな容量を保有しているので，かなり大きな 1 次元配列でも格納することができる．そのため後述するようにダイクストラ法の高速化においては特に重要性が高い (第 3.3，3.4，3.5 節参照)．

3.3. データアクセスパターンを考慮したデータ配置

データアクセスパターンの中長期予測が可能ならば，必要となるデータを高速なキャッシュメモリに予め移動させておくことでメモリ移動コストを隠蔽することが可能であるが，ダイクストラ法では動的計画法の特性から中長期的な予測が困難となり，データの再利用率も低い．そこで，局所的 (各点から隣接枝に対する走査) にアクセスパターンを考慮してデータ配

†仮想メモリアドレスと物理メモリアドレスの変換のためのバッファメモリ

©日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

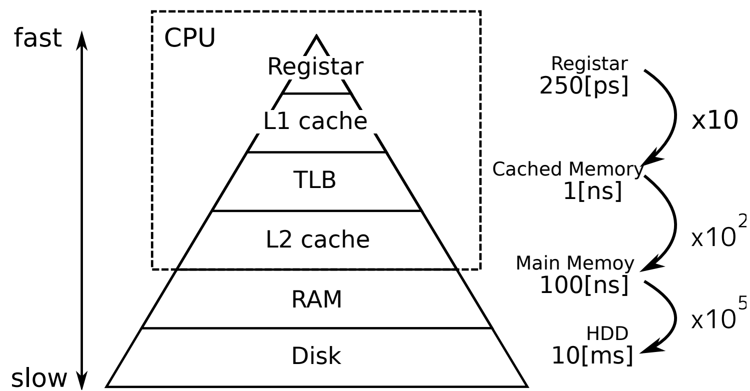


図 9: メモリ階層構造の例

置を選択することにより，データアクセスの密度を改善する．現在の一般的なプロセッサのキャッシュラインサイズは 64 bytes であるため，一度のデータアクセスで 64 bytes (32-bit 整数型 (4 bytes) であれば 16 要素) のデータをキャッシュ上に移動させることが可能である．全米道路ネットワークグラフでは各点から接続している枝は高々数本 (表 13 参照) であるため，多くの場合 1 度のデータアクセスで必要な枝情報が揃い 2 度目以降のメモリ移動コストを削減できる．

3.3.1. データアクセスパターンを考慮したグラフ表現の選択

点数 n，枝数 m のグラフを表現するために要求される領域は，隣接行列表現 (adjacency-matrix representation) では $O(n^2)$，隣接リスト表現 (adjacency-list representation) では $O(n+2m)$ となる [6]．道路ネットワークのように疎性の高いグラフに対し隣接行列表現を適用すると，各点に隣接する枝情報を参照するといったダイクストラ法の典型的なアクセスパターンには合致せず不連続なデータアクセスになってしまう (図 10 参照)．配列を用いて実装した隣接リスト表現を選択することで，各点に隣接する枝情報に対し連続的なデータアクセスを行うことが可能になる (図 11 参照)．

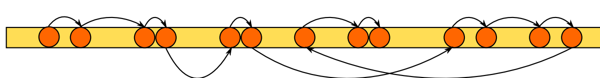


図 10: 不連続なデータアクセスパターン

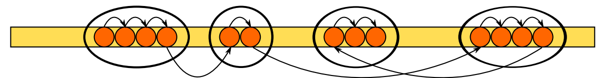


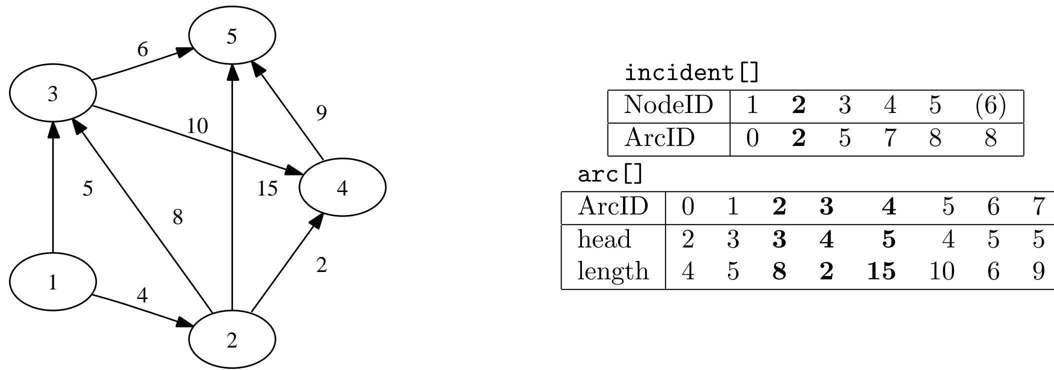
図 11: 局所的に連続なデータアクセスパターン

配列を用いて実装した隣接リスト表現は，始点番号から枝情報配列の要素番号へ変換するための配列 `incident[]` と，終点番号と枝の重みを要素にもつ枝情報配列 `arc[]` から構成される．枝情報配列は同一始点番号で連続的に配置しているため，点 v に隣接している枝番号は `incident[`v`]`，`incident[`v`+1]`，...，`incident[`v`+1]-1` となり，枝情報の参照は `arc[incident[`v`]]`，`arc[incident[`v`]+1]`，...，`arc[incident[`v`+1]-1]` と疎性によらず連続的である．図 12 のグラフでは，点 2(`incident[]` 配列の太字箇所) に接続されている枝情報は，枝 2 (終点 3，枝長 8)，3 (終点 4，枝長 2)，4 (終点 5，枝長 15) (`arc[]` 配列の太字箇所) となる．

3.3.2. データアクセスパターンを考慮したデータ配置による改善

枝情報配列 `arc[]` のデータ配置方法として，終点・枝の重みを要素とする構造体の配列を用いた実装 A_{struct} (図 13 参照) と，終点・枝の重みをそれぞれ個別の配列 `head[]`，`length[]` とした実装 A_{ary} (図 14 参照) の 2 種類が考えられる．要素間が密接に関連する場合 (同一の

©日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．



NodeID	1	**2**	3	4	5	(6)
ArcID	0	**2**	5	7	8	8

ArcID	0	1	**2**	**3**	**4**	5	6	7
head	2	3	**3**	**4**	**5**	4	5	5
length	4	5	**8**	**2**	**15**	10	6	9

図 12: 配列により実装された隣接リスト表現の例 (点数 5，枝数 8)

タイミング (もしくは短い期間) で要素を要求される) には A_{struct} が，要素間に関連性がない場合 (各要素はそれぞれ異なったタイミングで要求される) には A_{ary} を選択することが望ましい．ダイクストラ法に必要となる各点に対する作業領域や，優先キューに使用する領域に対しても，同様に 2 種類の配置方法 N_{struct}，N_{ary} が考えられる．

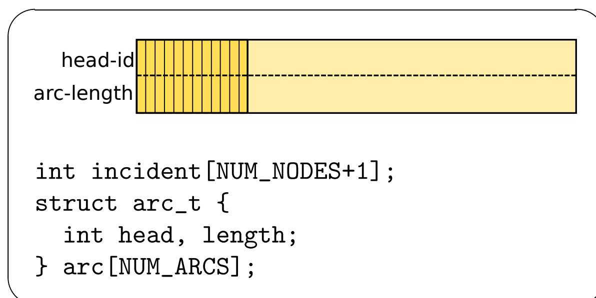


図 13: 構造体の配列による実装 A_{struct}

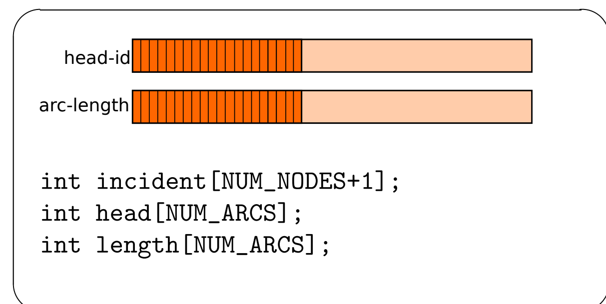


図 14: 個別の配列による実装 A_{ary}

図 15，表 3 は，グラフ表現とダイクストラ法の作業領域 (優先キュー) に対して前述した 2 種類の配置方法により実装されたダイクストラ法の性能をまとめたものである．各要素が個別の配列となるように配置した (A_{ary}, N_{ary}) を基準 (100%) とした性能比率となっている．計算機の仕様に関しては表 2 を参照されたい．計算機環境毎に性能特性は異なるものの，性能順は次のように一致している:

$$(A_{ary}, N_{ary}) \leq (A_{struct}, N_{ary}) \leq (A_{ary}, N_{struct}) \leq (A_{struct}, N_{struct}).$$

3.4. k-ary heap における k の適正値

図 16，表 4 は，全米道路ネットワークグラフ (表 13 参照) に対し k-ary heap における k パラメータを $k \in \{2, ..., 32\}$ と変化させた際の 1 対全最短路問題の実行時間をまとめたものである．先行研究 [4] では $k = 4$ を設定した 4-ary heap を用いることを推奨しているが，現在の計算機環境 Harpertown (表 2 参照) では $k = 2$ が適している．一般にヒープ構造における子の数 k が大きくなるに従い木の深さは低くなりスワップ操作の回数は減少するが，比較演算が増大するため性能低下すると考えられる．extract-min 操作では $O(k \log_k n)$ 回の比較演算を必要とする．

©日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

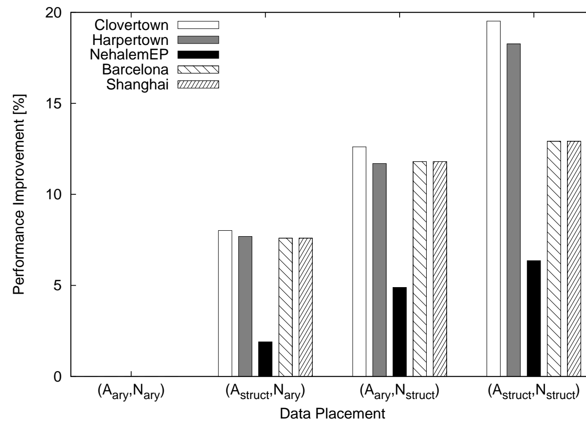


図 15: (A_{ary}, N_{ary}) を基準としたデータ配置毎の性能比率 [%]

表 3: (A_{ary}, N_{ary}) を基準としたデータ配置毎の性能比率 [%]

	(A_{ary}, N_{ary})	(A_{struct}, N_{ary})	(A_{ary}, N_{struct})	(A_{struct}, N_{struct})
Clovertown	0.0%	+8.02%	+12.61%	+19.52%
Harpertown	0.0%	+7.69%	+11.70%	+18.27%
Nehalem-EP	0.0%	+1.90%	+4.89%	+6.36%
Barcelona	0.0%	+7.59%	+11.80%	+12.92%
Shanghai	0.0%	+6.38%	+11.57%	+18.94%

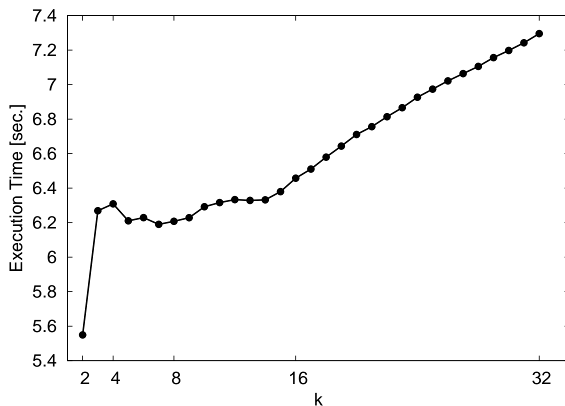


図 16: k-ary heap:k による実行時間の推移 [sec.]

表 4: k-ary heap:k による実行時間の推移 [sec.]

k	実行時間				
1-5	-	5.55	6.27	6.31	6.21
6-10	6.23	6.19	6.21	6.23	6.29
11-15	6.32	6.33	6.33	6.33	6.38
16-20	6.46	6.51	6.58	6.64	6.71
21-25	6.76	6.81	6.87	6.93	6.97
26-30	7.02	7.06	7.11	7.16	7.20
31-32	7.24	7.30	-	-	-

3.5. バイナリ・ヒープにおけるスワップ操作の改善

配列を用いて実装されたバイナリ・ヒープはデータの空間的局所性が非常に高い優先キューであるが，`insert`，`decrease-key`，`extract-min` のいずれの操作においても，ヒープ構造を維持するためには不連続なデータアクセスとスワップ操作が必要になり，この操作のために計算量から期待される性能を引き出すことは容易ではない．頻繁に不連続なデータアクセスやスワップ操作が行われると，直後に必要となるデータがキャッシュメモリ上から追い出される可能性が高まり，キャッシュミスによりメインメモリへのアクセス回数が増大してしまう．演算順序の入れ換えを行うことでスワップ操作 (**Algorithm 2**) をデータ移動 (**Algorithm 1**) に変換し，書き込み回数を約半数に削減することが可能である．`extract-min` を例に取り上げて説明をするが，`insert` や `decrease-key` に関しても同様に改善することができる．

バイナリ・ヒープ H の根に配置されたデータ $H[root]$ を取り除く処理について説明を行う．スワップ操作を用いた `topdown-extract-min`(H)(図 17，**Algorithm 1** 参照) では，末尾のデータを根に移動し，根から葉に向かいスワップ操作を繰り返す．一方，計算順序の入れ替えによりスワップ操作を排除した `bottomup-extract-min`(H)(図 18，**Algorithm 2**

©日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

参照) では，根に置かれるデータが末尾のデータによって押し出されるようにデータ移動を行う．表 5 は，`topdown-extract-min(`H`)` と `bottomup-extract-min(`H`)` を用いたバイナリ・ヒープを適用したダイクストラ法の実行時間をまとめたものである．全米道路ネットワークグラフ (表 13 参照) に対し，1 対 1 最短路問題を繰り返し計算した際の平均の実行時間である．いずれも 4 から 5%程度の改善が確認できる．

表 5: スワップ操作排除による性能向上の割合 [%]

	`topdown-extract-min(`H`)` [sec.]	`bottomup-extract-min(`H`)` [sec.]	性能向上
Clovertown	3.60	3.41	+5.52%
Harpertown	2.49	2.37	+4.78%
Nehalem-EP	2.56	2.41	+5.78%
Barcelona	4.47	4.30	+3.87%
Shanghai	3.76	3.58	+4.82%

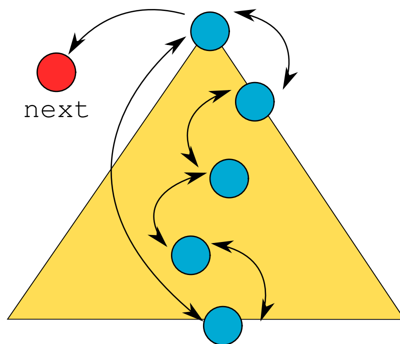


図 17: `topdown-extract-min`

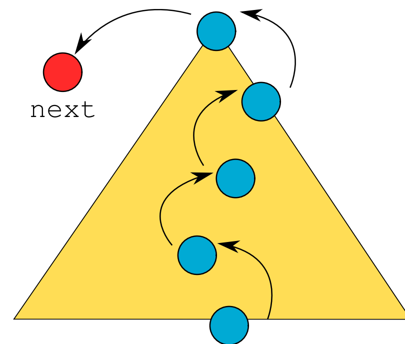


図 18: `bottomup-extract-min`

Algorithm 1 `topdown-extract-min(`H`)`

1: **if** $H = \emptyset$ **then**
2:     **return** nil
3: **end if**
4: $x \leftarrow H[root]$
5: $H[root] \leftarrow H[H.size]$
6: $i \leftarrow root$
7: **while** $H[left(i)] \neq \emptyset$ **do**
8:     **if** $H[left(i)].key < H[right(i)].key$ **or** $H[right(i)] = \emptyset$ **then**
9:         $min \leftarrow left(i)$
10:     **else**
11:         $min \leftarrow right(i)$
12:     **end if**
13:     **if** $H[min].key < H[i].key$ **then**
14:         $swap(H[i], H[min])$
15:         $i \leftarrow min$
16:     **else**
17:         **return** x
18:     **end if**
19: **end while**
20: **return** x

©日本オペレーションズ・リサーチ学会．無断複写・複製・転載を禁ず．

Algorithm 2 `bottomup-extract-min(H)`

1: **if** $H = \emptyset$ **then**
2:     **return** nil
3: **end if**
4: $x \leftarrow H[root]$
5: $i \leftarrow root$
6: **while** $H[left(i)] \neq \emptyset$ **do**
7:     **if** $H[left(i)].key < H[right(i)].key$ **or** $H[right(i)] = \emptyset$ **then**
8:         $min \leftarrow left(i)$
9:     **else**
10:         $min \leftarrow right(i)$
11:     **end if**
12:     **if** $H[min].key < H[H.size].key$ **then**
13:         $H[i] \leftarrow H[min]$
14:         $i \leftarrow min$
15:     **else**
16:         break
17:     **end if**
18: **end while**
19: **if** $H[i] \neq H[H.size]$ **then**
20:     $H[i] \leftarrow H[H.size]$
21: **end if**
22: **return** x

3.6. ダイクストラ法のクエリ並列化

ダイクストラ法は各反復における依存関係が非常に強く，並列実行向きのアルゴリズムとはいえない．そこで同一のグラフに対して最短路問題を繰り返し計算する際に，マルチコア・プロセッサ計算機資源を有効に利用するためのダイクストラ法に対するスレッド並列計算を提案する (**Algorithm 3**)．独立したダイクストラ法によってクエリを並列に処理するために，スレッド毎に作業領域が必要となる．また，グラフデータは全スレッドで共有する．本実装ではスレッド並列計算のために POSIX Pthreads ライブラリを用いる．しかしながらダイクストラ法は一般的にメモリ帯域幅に律速するアルゴリズムであるため，単純なジョブ並列であっても期待通りの性能向上することは困難である．そのため第 3.3，3.4，3.5 節で示したメモリ帯域幅の負荷を軽減する工夫が重要になってくる．

Algorithm 3 ダイクストラ法のクエリ並列実行

input: 始終点集合 S
1: *generate all threads*
2: **while** $S \neq \emptyset$ **do**
3:     $(s, t) \leftarrow pop(S)$ *with mutex*
4:     $dijkstra(s, t)$
5: **end while**
6: *terminate all threads*

3.7. その他の改善

- より小さなデータ型を選択し，メモリ要求量やデータ移動量を抑える．
- 余算などの一般的に演算コストの大きな演算をコストの小さな演算へ変換する．

ⓒ日本オペレーションズ・リサーチ学会．　無断複写・複製・転載を禁ず．

- 不要なアドレス計算やデータ移動を削減する.

4. メモリ階層構造上の律速箇所の解析

次にメモリ階層構造を考慮し，汎用的な評価を行うための実験方法を示す．プロファイラなどの性能解析ソフトウェアによる律速 (ボトルネック) 箇所の限定は非常に有効であるが，微小区間の計測などプロファイラ自身のオーバーヘッドにより正しく測定できない場合も少なくない．また，詳細なプロファイル結果を得られても計算機のどの箇所に律速しているか判断することは容易ではない．そのためプロファイラ等を用いずに複数の計算機環境 (表 2 参照) に対し計算機実験を行うことで，メモリ階層構造上のどの箇所に律速しているかを把握していく．本実験で用いる計算機はいずれもクアッドコア・プロセッサを 2 基搭載した 8 プロセッサコア環境であるが，計算機環境 Clovertown，Harpertown (図 19 参照) は L2 キャッシュメモリを 2 プロセッサコアで共有しているのに対し，計算機環境 Nehalem-EP (図 20 参照)，Barcelona，Shanghai は L2 キャッシュメモリは各プロセッサコア毎に置かれ，新たに 4 プロセッサコア共通の L3 キャッシュメモリが配置されている．なお，コンパイラは GNU GCC(gcc-4.1.2，gcc-4.4.3)，最適化オプションは-O2 を用いる (表 6 参照).

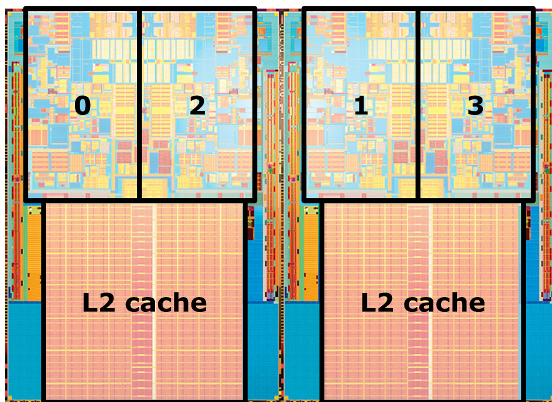


図 19: Intel Xeon(R) X5460: 共有型 L2 キャッシュメモリ・クアッドコア・プロセッサ

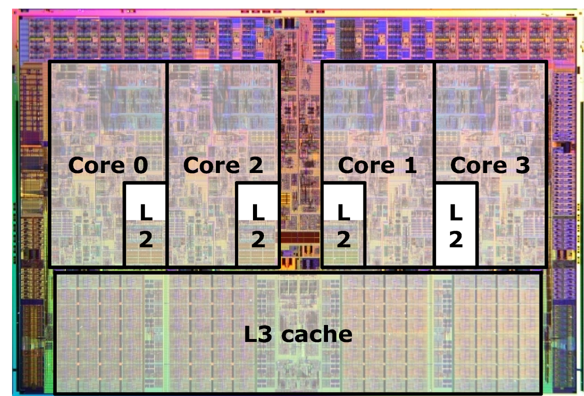


図 20: Intel Xeon(R) X5550: 非共有型 L2 キャッシュメモリ・クアッドコア・プロセッサ

表 6: コンパイラと最適化オプション

コンパイラ	gcc-4.1.2，gcc-4.4.3
最適化オプション	-O2

4.1. プロセッサの動作周波数を変化させて実行

まず初めにメモリ帯域幅を固定しプロセッサの動作周波数のみ変化させ，その際の実行時間の変化率からメモリ帯域幅に律速されているか判定する．プロセッサの動作周波数の変更に Linux 上の `cpufreq-selector` コマンドを用いる．動作周波数を低下させることで演算処理能力は低下するが，メモリコントローラの周波数はプロセッサコアの周波数とは独立であるため，メインメモリ $\Longleftrightarrow$ プロセッサ間のメモリ帯域幅は変化しない．動作周波数の低下に従い実行時間が変化すると演算量に律速し，実行時間が変化しない場合や変化量が非常に小さい場合にはデータ移動量に律速するといえる (表 7 参照)．動作周波数が低下すると演算に対するデータ移動量の割合は小さくなる．動作周波数を大きく変化させると，律速箇

©日本オペレーションズ・リサーチ学会．無断複写・複製・転載を禁ず．

所はメモリ帯域幅から演算処理能力へ逆転する．Intel Xeon(R) 5160 3.00GHz の動作周波数を 2.00GHz (−33%) に低下させ，その際の 1 対全最短路問題の計算時間の差より 27.01% の性能低下が確認されるため，メモリ帯域幅以外に律速すると判断できる (表 8 参照).

表 7: プロセッサの動作周波数変化実験による律速箇所の限定

実行時間	律速箇所
変化なし	メモリ帯域幅に律速
変化あり	メモリ帯域幅以外に律速

表 8: プロセッサの動作周波数変化による実行時間の変化量 [%]

	変化前の動作周波数	変化後の周波数 (変化量)	性能比率
Intel Xeon(R) 5160	3.00GHz	2.00GHz (−33 %)	**−27.01**%

4.2. 2 プロセッサコア同時実行

次に 1 プロセス実行性能に対し，特定のプロセッサコアの組合せ (表 9 参照) による 2 プロセス同時実行性能の変化率 (低下率) から，メモリ階層構造上の律速箇所を特定する (表 10 参照)．プロセッサコア指定に Linux 上の `numactl` コマンドを用いる．同時に実行させる 2 プロセスは同一のグラフ (全米道路ネットワークグラフ，表 13 参照) を用いて同一のクエリ (1 対全最短路問題 1 回) を計算する．その際メモリアクセスは 2 プロセス間で独立となりキャッシュメモリによる効率化も行われないため，プロセス間で共有されるメモリ帯域幅やキャッシュメモリへの負荷が高まる．つまり本実験では，より厳しい資源制約による性能低下率から律速箇所の特定を行っている．2 プロセス分のメモリ帯域幅に律速するならば，異なるソケットに配置されているコアの組合せでは性能に変化はないが，同一ソケットに配置されているコアの組合せでは性能低下が起きてしまう．

表 9: クアッドコア・プロセッサにおける 2 プロセッサコアの組合せ

コアの組合せ	詳細
1 コア	基準とする実行時間
異なるソケット	異なるソケット上の 2 コア
L2 非共有コア	同一ソケット上の L2 キュッシュメモリを共有しない 2 コア (例: 図 19 の 0 番と 1 番)
L2 共有コア	同一ソケット上の L2 キュッシュメモリを共有する 2 コア (例: 図 19 の 0 番と 2 番)

表 10: 1 プロセス実行性能に対する 2 プロセス同時実行による性能変化 (低下) と律速箇所の関係

律速箇所	異なるソケット	L2 非共有 2 コア	L2 共有 2 コア
1 プロセス分のメモリ帯域幅	変化あり	変化あり	変化あり
2 プロセス分のメモリ帯域幅	変化なし	変化あり	変化あり
L2 キャッシュメモリの共有	変化なし	変化なし	変化あり
演算性能	変化なし	変化なし	変化なし

まず，計算機環境 Harpertown (Intel Xeon(R) X5460，表 2，図 19 参照) において，優先キュー毎の実験結果をまとめる (表 11 参照)．特にバイナリ・ヒープを適用したダイクストラ法 2-heap* (優先キューに使用する領域を動的確保) は，他の優先キューとの実装の中でも最も性能低下が小さい．

次に 2-heap* の計算機環境毎の実験結果を表 12 にまとめる．表 12 から，主な律速箇所はメモリ帯域幅ではなく，L2 キャッシュメモリの共有による性能低下であると判定される．つまりスレッド並列計算において L2 キャッシュメモリを共有しないように各スレッドをプ

ⓒ日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

表 11: 2 プロセッサコア同時実行による優先キュー毎の性能比率 [%](実行時間 [sec.])

	1 コア	異なるソケット	L2 非共有 2 コア	L2 共有 2 コア
2-heap*	0.00% (5.46)	−1.05% (5.52)	−4.03% (5.69)	−18.56% (6.70)
2-heap	0.00% (5.67)	−1.59% (5.82)	−6.33% (6.17)	−19.67% (7.12)
buckets	0.00% (3.93)	−4.34% (4.13)	−12.48% (4.56)	−30.27% (5.60)
mbp	0.00% (5.69)	−2.00% (5.85)	−6.72% (6.17)	−26.90% (7.73)

ロセッサコアに割り振ることで，並列効率が非常に高い並列計算が可能になることを示している．スレッド並列計算では，他プロセッサコアと資源を奪い合うことなく実行できるアルゴリズムを選択することが非常に重要である．

表 12: 2 プロセッサコア同時実行による計算機環境毎の性能比率 [%](実行時間 [sec.])

	1 コア	異なるソケット	L2 非共有 2 コア	L2 共有 2 コア
Clovertown	0.00% (7.65)	+1.02% (7.58)	−4.24%(7.99)	−22.21% (9.84)
Harpertown	0.00% (5.46)	−1.05% (5.52)	−4.03%(5.69)	−18.56% (6.70)
Nehalem-EP	0.00% (5.51)	+0.53% (5.49)	−3.13%(5.69)	-(-)
Barcelona	0.00% (9.80)	+0.02% (9.80)	−5.81%(10.40)	-(-)
Shanghai	0.00% (8.32)	+3.91% (8.01)	−5.66%(8.82)	-(-)

4.3. 2 プロセッサコア同時読込み

次に L2 キャッシュメモリを共有することによる性能低下の要因に関して詳細を解析していく．各プロセッサコア毎に独立したダイクストラ法を割り当てるスレッド並列計算 (**Algorithm 3**) において，キャッシュメモリを共有したマルチコアプロセッサ計算機環境 Clovertown，Harpertown (図 19 参照) では，各プロセッサコアが使用できるキャッシュメモリ領域は共有した数だけ小さくなる．そのため，一方のプロセッサコアがキャッシュメモリ上に配置したデータを，もう一方のプロセッサコアが取得したデータによって押し出されるといった資源の競合が予測される．同様にメインメモリ $\Longleftrightarrow$ プロセッサコア間のメモリ帯域幅も相対的に狭くなるが，第 4.2 節で示した実験結果 (表 11) により律速箇所ではないと判断されている．2 プロセッサコアから同時に同一領域に対するデータ参照を行った際のメモリ帯域幅 (図 21 参照) と，1 プロセッサコア実行に対する 2 プロセッサコア同時実行による性能低下率 (表 12 参照) を比較することで，キャッシュメモリ資源の競合による影響を確認する．

図 21 は，Harpertown (Intel Xeon X5460 3.16GHz (図 19 参照)$\times$2) 上で参照するデータサイズを変化させながらメモリ帯域幅を計測したものである．L2 キャッシュメモリを参照しているデータサイズ (64Kbytes から 6Mbytes) では，L2 キャッシュメモリを共有したコアの組合せは他の場合に比べ 16% 程の性能低下が確認される (図 21 参照)．この割合は 2 プロセッサコア同時実行による性能低下率 (−18.56%) と同程度であるため，L2 キャッシュメモリの共有による資源の競合による影響は小さく，プロセッサコア $\Longleftrightarrow$L2 キャッシュメモリ間の帯域幅に律速されたと判断できる．

5. メモリ階層構造による高速化後の性能

最後に 9th DIMACS ベンチマーク問題から最大の規模である全米道路ネットワークグラフ (表 13 参照) を用いて性能比較実験を行う．

5.1. 1 対全最短路問題に対する優先キュー毎のダイクストラ法の性能

全米道路ネットワークグラフ (表 13 参照) に対し 1 対全最短路問題の実行時間を示す．本研究で高速化を行った 2-heap，2-heap* (優先キュー領域を最大サイズ (点数 n) で予め確保せ

©日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

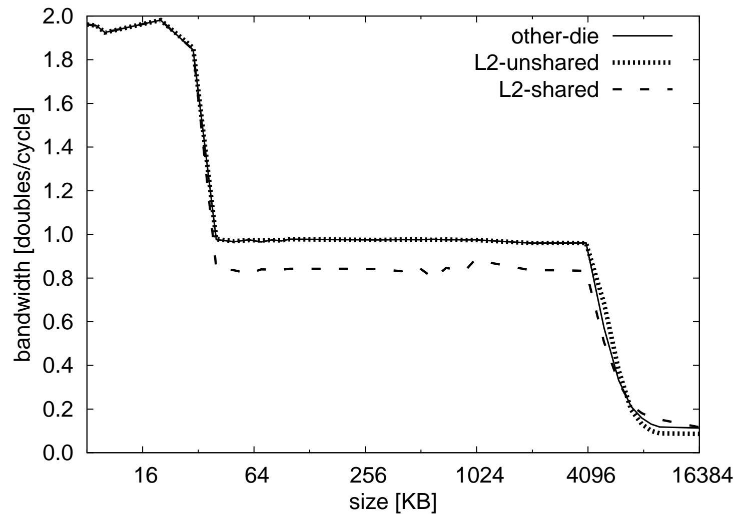


図 21: 2 プロセッサコア同時読込みにおけるメモリ帯域幅 [doubles/cycle]

表 13: 全米道路ネットワークグラフ

点数	23,947,347	交差点の数 n
枝数	58,333,344	交差点間の道路の数 m
枝長	[1, 368855]	枝長の範囲: 最大枝長 C
次数	[1, 8]	点に接続している枝数の範囲

ず格納数に応じて拡張)，buckets 以外に，9th DIMACS でベンチマーク・ソルバーとして使用された mbp[14] を始め，複数の実装 [4, 5, 16] に対しての比較実験結果を，表 14 (本研究，[14])，表 15 ([4,16])，表 16 ([5]) にまとめる．Dijk は始点を 1 とする 1 対全最短路問題の計算時間 (単位は秒)，Init はグラフデータ読込み時間 (単位は秒)，Mem はメモリ要求量 (単位は GByte) をそれぞれ表している．なお，計算機環境は Harpertown (表 2 参照) である．

本研究で開発した 2-heap，2-heap*，buckets (表 14 内の太字) はメモリ要求量を抑えながら高速な実行が可能である．また，大規模グラフデータを，予めバイナリデータに変換しておくことで，他のソルバーに比べ極めて高速な初期化時間で実行が可能となる．バイナリデータの作成に要する時間は数十秒となっている．

表 14: 性能比較実験 1

	計算量	アルゴリズム	Dijk[sec.]	Init[sec.]		Mem[GB]
				text-file	binary-file	
2-heap*	$O(m \log_2 n)$	binary heap	**5.53**	**30.30**	**0.91**	**0.90**
2-heap	$O(m \log_2 n)$	binary heap	**5.60**	**30.30**	**0.91**	**1.27**
buckets	$O(m + nC)$	Dial's algorithm	**3.50**	**30.30**	**0.91**	**0.91**
mbp[14]	$O(m + n \log C)$	multi-level buckets	5.68	51.70	-	2.17

5.2. データ型による性能比率

最大の全米道路ネットワークグラフ (表 13 参照) においても，点数・枝長は 32-bit 整数型で表現可能である．2-heap，2-heap*，buckets では，探索前にデータ型の確定が困難な距離ラベル変数は 64-bit 整数型とし，それ以外の変数に関しグラフデータに応じてより小さなデータ型を自動的に選択するよう実装している．より小さなデータ型を選択したことによる性能向上やメモリ要求量の軽減を確認できる (表 17 と表 18 (1 対 1 最短路問題の 1 クエリあたりの実行時間) 参照)．一方 mbp はすべての変数で 64-bit 整数型を選択しており，

©日本オペレーションズ・リサーチ学会．無断複写・複製・転載を禁ず．

表 15: 性能比較実験 2 [4]

	計算量	アルゴリズム	Dijk[sec.]	Init[sec.]	Mem[GB]
DIKQ	$O(n^2)$	naive Dijkstra's algorithm	674.13	51.28	1.81
DIKH	$O(m \log_k n)$	k-ary heap ($k = 4$)	7.23	51.94	2.43
DIKR	$O(m + n \log C)$	1-level R-heap($\log_2 C + 3$)	8.09	52.87	2.80
DIKF	$O(m + n \log n)$	fibonacci heap	15.97	53.41	3.17
DIKB	$O(m + nC)$	Dial's algorithm	4.38	50.38	2.62
DIKBD	$O(m + n(\Delta + C/\Delta))$	double buckets $\Delta = \lceil C/2^{11} \rceil$	4.67	52.59	2.62

表 16: 性能比較実験 3 [5, 16]

	計算量	アルゴリズム	Dijk[sec.]	Init[sec.]	Mem[GB]
b1	$O(m + n(C))$	1-level buckets	5.58	52.48	2.55
b2	$O(m + n(C^{1/2}))$	2-level buckets	5.57	52.51	2.55
b3	$O(m + n(C^{1/3}))$	3-level buckets	5.59	52.35	2.55
h1	$O(m + n(\log C)^{(1/4)+\epsilon})$	1-level hot queue	5.93	52.39	2.55
h2	$O(m + n(\log C)^{(1/4)+\epsilon})$	2-level hot queue	5.93	52.47	2.55
h3	$O(m + n(\log C)^{(1/4)+\epsilon})$	3-level hot queue	5.92	52.46	2.55

32-bit 整数に変更しても優先キューの特性から性能向上は見込めない．なお，計算機環境は Harpertown (表 2 参照) である．

表 17: データ型による実行時間 [sec./query]

	2-heap*	2-heap	buckets	mbp
32-bit 整数型	2.61	2.47	1.68	2.63
64-bit 整数型	3.11	3.14	2.28	2.65

表 18: データ型によるメモリ要求量 [GB]

	2-heap*	2-heap	buckets	mbp
32-bit 整数型	0.90	1.27	1.09	2.17
64-bit 整数型	1.62	1.99	1.81	2.17

5.3. 全米道路ネットワークグラフに対するバイナリ・ヒープを適用したダイクストラ法

表 19，図 22 は，2-heap の計算機環境毎 (表 2 参照) の 1 対 1 最短路問題 1 クエリあたりの実行時間をまとめたものである．L2 キャッシュメモリを 2 プロセッサコアで共有するクアッドコア・プロセッサ (計算機環境 Clovertown，Harpertown) 上での 8 スレッド並列時の性能は，4 スレッド並列までと比べて効率が低下しているが，プロセッサコア毎に L2 キャッシュメモリを保持しているクアッドコア・プロセッサ (計算機環境 Nehalem-EP，Barcelona, Shanghai) では 8 スレッド並列時においても大きな性能低下は確認されず並列効率が非常に高い．

5.4. 1 対 1 最短路問題に対する優先キュー毎のダイクストラ法の性能

メモリ階層構造を考慮した高速化を行った 2-heap，2-heap*，buckets は，いずれもメモリ使用量が少なくスレッド並列計算による並列効率が非常に高い (図 23， 24，表 20， 21 参照)．特に 2-heap*(4 スレッド並列) は mbp と比べて同量のメモリ要求量で 4.02 倍高速である (表 20， 21 内の太字)．2 プロセッサコア同時実行により性能低下 (表 11 参照) が 2-heap や 2-heap* に比べて大きな mbp は，スレッド並列計算による並列効率が 2-heap や 2-heap* に比べ低いと予想される．他の道路ネットワークグラフ (表 22 参照)，ベンチマーク問題 (表 1 参照) に対しても同等の高速化を確認できる (Appendix A，B，C，D，E，F，G)．なお，計算機環境は Harpertown(表 2 参照) である．

6. 最短路問題オンライン・ソルバー

2008 年から著者らのグループでは 9th DIMACS で公開されている道路ネットワークグラフを用いた大規模最短路問題を，ウェブブラウザによる GUI (グラフィカル・ユーザ・イン

©日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

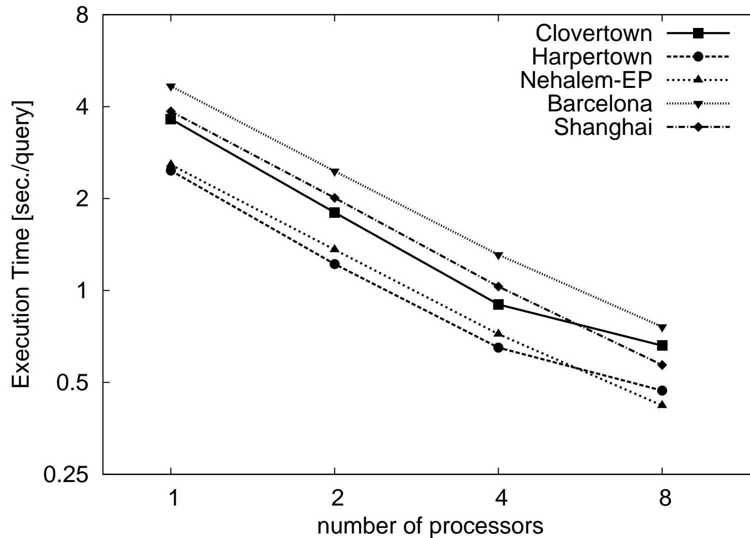


図 22: 計算機環境毎のダイクストラ法の実行時間 [sec./query]

表 19: 計算機環境毎のダイクストラ法の実行時間 [sec./query]

number of processors	1	2	4	8
Clovertown	3.64	1.80	0.90	0.66
Harpertown	2.47	1.22	0.65	0.47
Nehalem-EP	2.58	1.36	0.72	0.42
Barcelona	4.67	2.46	1.31	0.76
Shanghai	3.86	2.01	1.03	0.57

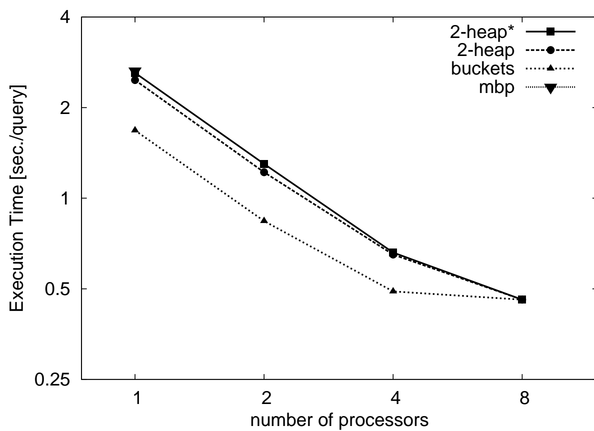


図 23: 優先キュー毎の実行時間 [sec./query]

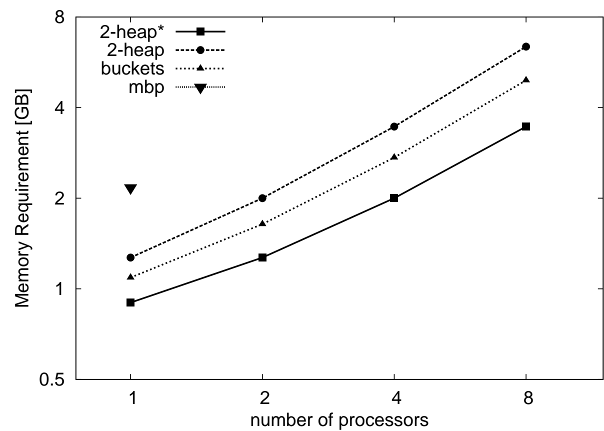


図 24: 優先キュー毎のメモリ要求量 [GB]

表 20: 優先キュー毎の実行時間 [sec./query]

number of processors	1	2	4	8
2-heap*	2.61	1.30	**0.66**	0.46
2-heap	2.47	1.22	0.65	0.46
buckets	1.68	0.84	0.49	0.46
mbp	**2.65**	-	-	-

表 21: 優先キュー毎のメモリ要求量 [GB]

number of processors	1	2	4	8
2-heap*	0.90	1.27	**2.00**	3.46
2-heap	1.27	2.00	3.46	6.38
buckets	1.09	1.64	2.73	4.93
mbp	**2.17**	-	-	-

ターフェイス) で利用可能な最短路問題オンライン・ソルバー‡として開発，公開している (図 25，26 参照)．本システムで用いた最短路問題ソルバーは，本研究で開発した前処理を

‡http://opt.indsys.chuo-u.ac.jp/portal/

©日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

行わずに厳密解を計算するダイクストラ法である．グラフデータを予めメモリ上に配置していないため，本研究で開発した最短路問題ソルバーの性能を体験することが可能である．ネットワークと表示に要する時間を除けば，実行時間は非常に高速である．

6.1. 最短路問題オンライン・ソルバーのシステム概要

本システムは図 27 のように複数の階層に分かれている．ユーザは，一般的な Web ブラウザからオンライン・ソルバーにアクセスし，マウスなどのポインティング・デバイスを用いて，始点や終点 (経路点を選ぶことの可能な種類も存在する) を指定する．計算サーバは，ユーザから指定された最短路問題を計算し，結果を経路ファイルもしくは画像ファイルを出力し，それを元に表示する．特に最短路問題ソルバーによる計算は，フロントエンドサーバでなくてもよく，クラスタ・コンピューティングやクラウド・コンピューティングとの連携により，さらに大規模な問題に対応することが可能になる．

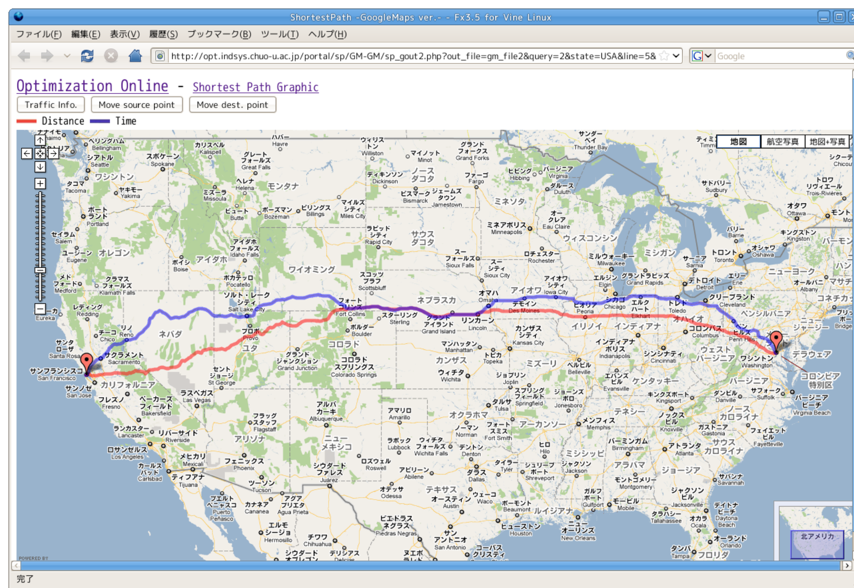


図 25: 最短路問題オンライン・ソルバーの実行画面:移動距離と所要時間による最短路問題

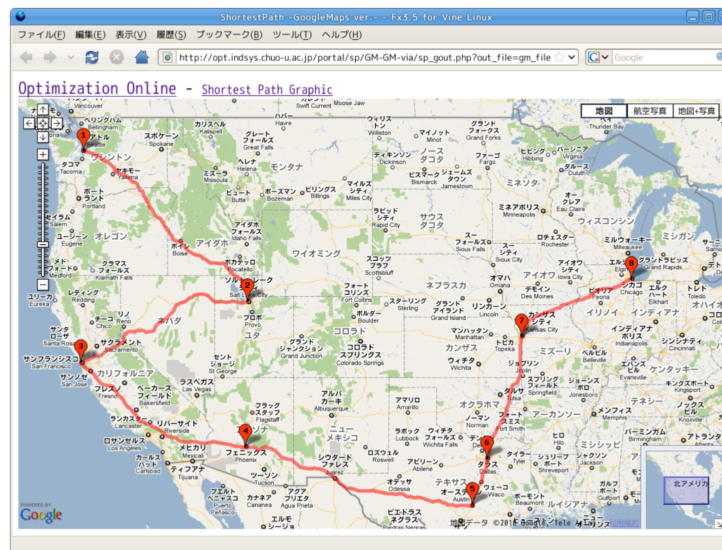


図 26: 最短路問題オンライン・ソルバーの実行画面:経由点あり最短路問題

©日本オペレーションズ・リサーチ学会．無断複写・複製・転載を禁ず．

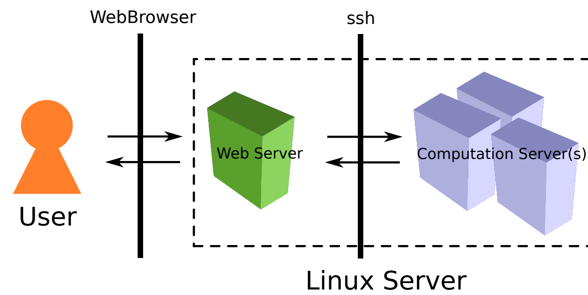


図 27: 最短路問題オンライン・ソルバーのシステム概要

7. おわりに

本研究ではメモリ階層構造を考慮することにより，大規模最短路問題に対するダイクストラ法に対して汎用的かつ大幅な高速化を行うための実装方法を示した．本研究で高速化を施したバイナリ・ヒープを適用したダイクストラ法は，メモリ要求量を抑えながらマルチコア・プロセッサ計算機性能を引き出すことが可能である．先行研究のマルチレベル・バケット (スレッド並列計算には非対応) に対し，1 スレッド計算時にはメモリ要求量は半分程度で同程度の性能を，4 スレッド計算時にはメモリ要求量は同量に抑えつつ 4.02 倍高速となる．

また，数値実験により本実装は L2 キャッシュメモリ帯域幅への律速が確認され，L2 キャッシュメモリを共有しないプロセッサコアの組合せでは，非常に効率的なスレッド並列計算が可能である．グラフ特性に対するメモリ要求量や実行時間の安定性，省メモリ性，マルチコア・プロセッサ環境での性能など，総合的に評価すると最も優れているといえる．本研究で扱った実装方法，評価手法は非常に汎用的であり，ダイクストラ法や優先キューだけに限定せずにアルゴリズム実装に広く適用可能で高速化が期待できる．

また，最短路問題オンライン・ソルバーとしてウェブ上に公開しているため，ウェブブラウザによる GUI 操作で容易にソルバーの性能を確認可能である．非常に大規模な実道路ネットワークグラフ (約 2400 万点，約 5800 万枝) に対し，リアルタイムかつ高速計算 (数秒) を行うシステムは類を見ない．

8. 今後の展開

現在，経路探索システムは世界中で広く利用されているが，計算機資源制約の厳しい小さな機器上で高速に探索しなければならないために，あまり多くの機能を搭載することができないのが現状である．そこで，集約された高性能並列計算サーバにより，大規模災害時における避難経路など，渋滞・事故など動的情報を踏まえて経路探索を行うことができる次世代経路探索システムの構築を目指している．さらには，得られた結果を用いて各ユーザに渋滞を避けるような経路を返すというような交通管制を行うことも模索していきたいと考える．

参考文献

[1] 9th DIMACS Implementation Challenge. `http://www.dis.uniroma1.it/~challenge9/`.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman: *The Design and Analysis of Computer Algorithms* (Addison-Wesley, 1974).

[3] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes: In transit to constant time shortest-path queries in road networks. In *SIAM Workshop on Algorithm Engineering*

ⓒ日本オペレーションズ・リサーチ学会．無断複写・複製・転載を禁ず．

and Experiments (ALENEX '07) (2007), 46–59.

[4] B.V. Cherkassky, A.V. Goldberg, and T. Radzik: Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, **73** (1996), 129–174.

[5] B.V. Cherkassky, A.V. Goldberg, and C. Silverstein: Buckets, heaps, lists, and monotone priority queues. *SIAM Journal on Computing*, **28** (1999), 1326–1346.

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein: *Introduction to Algorithms*, Second edition (MIT Press, 2001).

[7] D. Delling, P. Sanders, D. Schultes, and D. Wagner: Highway hierarchies star. *9th DIMACS Implementation Challenge* (2006).

[8] C. Demetrescu, A.V. Goldberg, and D.S. Johnson: 9th DIMACS implementation challenge benchmark solvers. *9th DIMACS Implementation Challenge* (2006).

[9] C. Demetrescu, A.V. Goldberg, and D.S. Johnson: 9th DIMACS implementation challenge core problem families. *9th DIMACS Implementation Challenge* (2006).

[10] R.B. Dial: Algorithm 360: Shortest path forest with topological ordering. *Communications of the ACM*, **12-11** (1969), 632–633.

[11] E.W. Dijkstra: A note on two problems in connexion with graphs. *Numerische Mathematik*, **1-1** (1959), 269–271.

[12] M.L. Fredman and R.E. Tarjan: Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, **34-3** (1987), 596–615.

[13] J.L. Hennessy and D.A. Patterson: *Computer Architecture: A Quantitative Approach*, Fourth edition (Morgan Kaufmann, 2006).

[14] A.V. Goldberg: A simple shortest path algorithm with linear average time. In *Algorithms—ESA 2001, 9th Annual European Symposium*, *Lecture Notes in Computer Science*, **2161** (2001), 230–241.

[15] A.V. Goldberg, H. Kaplan, and R.F. Werneck: Reach for A*: Efficient point-to-point shortest path algorithms. In *SIAM Workshop on Algorithm Engineering and Experiments (ALENEX '06)* (2006).

[16] A.V. Goldberg and C. Silverstein: Implementations of Dijkstra's algorithm based on multi-level buckets. *NEC Research Institute Technical Report*, **95-187** (1995).

[17] K. Goto and R.A. van de Geijn: Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, **34-3** (2008).

[18] K. Goto and R.A. van de Geijn: High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software*, **35-1** (2008).

[19] GotoBLAS. `http://www.tacc.utexas.edu/resources/software/`.

[20] J.W.J. Williams: Algorithm 232: Heapsort. *Communications of the ACM*, **7-6** (1964), 347–348.

©日本オペレーションズ・リサーチ学会. 無断複写・複製・転載を禁ず.

Appendix. 9th DIMACS ベンチマーク問題

9th DIMACS ベンチマーク問題 (表 1，22 参照) に対し，本研究で高速化を行ったバイナリ・ヒープと 1 レベル・バケットをそれぞれ適用したダイクストラ法 2-heap，buckets と，ベンチマーク・ソルバーであるマルチレベル・バケットを適用したダイクストラ法 mbp を用いた実験結果を示す．また，2-heap* は探索に応じて優先キューに使用する配列を動的に拡張するバイナリ・ヒープを適用したダイクストラ法である．なお計算機環境は，Harpertown (図 19，表 2 参照) を用いた．各表の実行時間はランダムに始点と終点を生成した 1 対 1 最短路問題 1000 クエリの平均実行時間 (msec.:ミリ秒) である．

A. USA-road-d Family

表 22: 道路ネットワーク・グラフの特性

	略称	点数	枝数	枝長
USA-road-d.USA.gr	USA	23,947,347	58,333,344	[1, 368,855]
USA-road-d.CTR.gr	CTR	14,081,816	34,292,496	[1, 214,013]
USA-road-d.W.gr	W	6,262,104	15,248,146	[1, 368,855]
USA-road-d.E.gr	E	3,598,623	8,778,114	[1, 200,760]
USA-road-d.LKS.gr	LKS	2,758,119	6,885,658	[1, 138,911]
USA-road-d.CAL.gr	CAL	1,890,815	4,657,742	[1, 215,354]
USA-road-d.NE.gr	NE	1,524,453	3,897,636	[1, 63,247]
USA-road-d.NW.gr	NW	1,207,945	2,840,208	[1, 128,569]
USA-road-d.FLA.gr	FLA	1,070,376	2,712,798	[1, 214,013]
USA-road-d.COL.gr	COL	435,666	1,057,066	[1, 137,384]
USA-road-d.BAY.gr	BAY	321,270	800,172	[1, 94,305]
USA-road-d.NY.gr	NY	264,346	733,846	[1, 36,946]

A.1. 道路ネットワーク・グラフに対する枝長変化実験

図 28 は，9th DIMACS で公開されている道路ネットワーク・グラフ USA-road-d.NY.gr (表 22 参照) に対し，横軸パラメータの $t \in \{-10, ..., 10\}$ により枝長に対しスケーリング (枝長 $l(e)$ を新たに $l'(e) = l(e) \times 2^t$ $(\forall e \in E)$ とする) を行い，枝長変化による優先キューの実行時間をまとめたものである．なお点数，枝数，枝の接続情報を固定している．

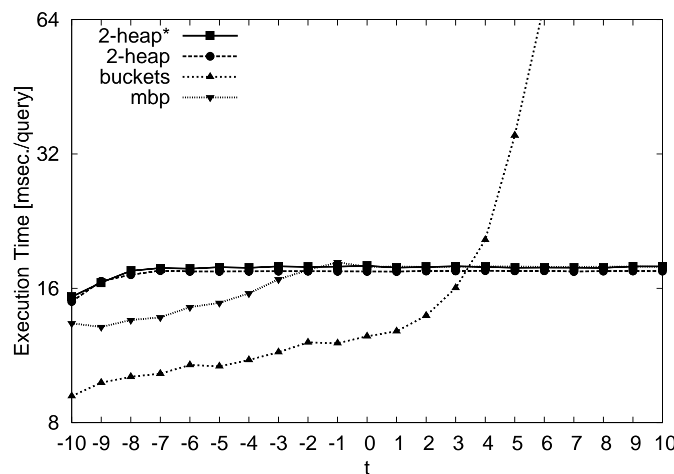


図 28: 枝長スケーリングによる実行時間の推移 [msec./query]

©日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

A.2. 大規模道路ネットワーク・グラフに対する 1 対 1 最短路問題

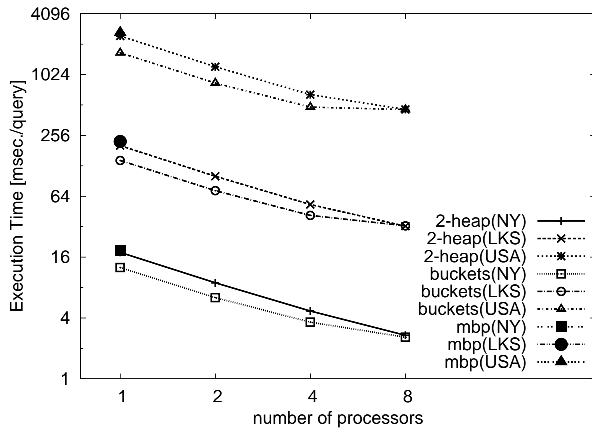


図 29: Execution Time[msec./query]

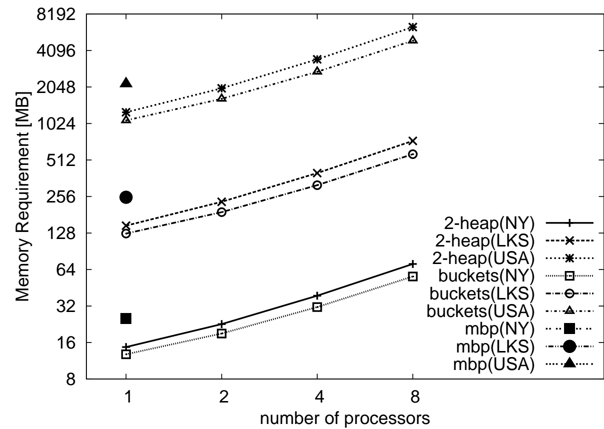


図 30: Memory Requirement[MB]

表 23: Execution Time[msec./query]

	2-heap				buckets				mbp
number of processors	1	2	4	8	1	2	4	8	
NY	17.8	8.9	4.7	2.7	12.6	6.4	3.6	2.6	18.5
BAY	19.4	9.7	5.1	3.1	14.6	7.4	4.3	3.2	20.7
COL	29.2	14.6	7.6	4.5	22.7	11.4	6.4	4.8	31.6
FLA	73.2	36.4	19.3	11.6	54.3	29.5	15.7	12.5	77.2
NW	90.3	44.9	23.6	14.1	64.0	32.2	18.3	14.1	95.0
NE	110.2	55.1	28.8	17.5	76.2	38.4	22.0	16.8	113.2
CAL	138.8	69.1	36.6	22.4	102.3	51.6	29.6	23.9	146.4
LKS	202.8	101.0	53.1	32.4	144.8	72.8	41.4	32.7	222.8
E	293.6	145.8	77.2	48.1	206.8	103.8	59.6	48.9	303.1
W	512.1	254.1	135.2	85.6	374.3	188.6	109.0	90.6	557.5
CTR	2052.5	1023.3	549.2	436.1	1535.8	780.1	455.2	459.0	2207.9
USA	2467.9	1224.3	648.6	462.0	1680.4	844.9	487.6	461.4	2649.3

表 24: Memory Requirement[MB]

	2-heap				buckets				mbp
number of processors	1	2	4	8	1	2	4	8	
NY	15	23	39	71	13	19	31	56	25
BAY	17	27	47	86	15	23	38	69	29
COL	23	36	63	116	20	31	52	94	39
FLA	57	90	155	286	50	75	126	227	99
NW	63	100	174	321	54	83	139	251	108
NE	82	129	222	408	71	106	176	317	141
CAL	101	158	274	504	87	131	219	396	172
LKS	147	231	400	736	127	190	318	572	252
E	191	300	520	959	164	247	413	746	326
W	331	522	905	1669	285	430	719	1298	567
CTR	745	1175	2034	3753	639	962	1608	2900	1275
USA	1267	1998	3460	6383	1086	1635	2735	4933	2169

©日本オペレーションズ・リサーチ学会. 無断複写・複製・転載を禁ず.

B. Long-n Family に対する 1 対 1 最短路問題

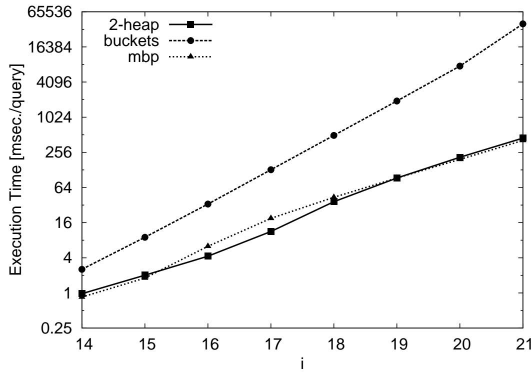


図 31: Execution Time[msec./query]

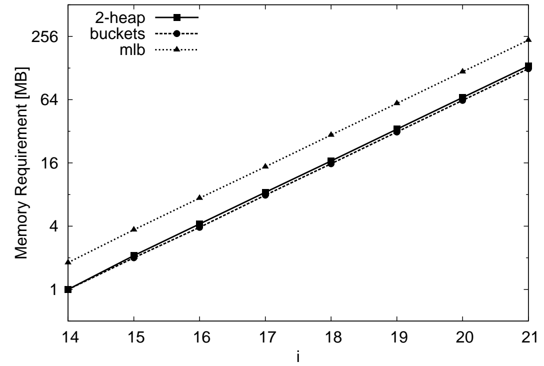


図 32: Memory Requirement[MB]

表 25: Execution Time[msec./query]

	2-heap	buckets	mbp
Long-n.14.0.gr	1.0	2.5	0.9
Long-n.15.0.gr	2.0	9.0	1.8
Long-n.16.0.gr	4.3	33.3	6.3
Long-n.17.0.gr	11.3	128.9	18.9
Long-n.18.0.gr	36.6	499.0	43.4
Long-n.19.0.gr	93.1	1937.6	93.0
Long-n.20.0.gr	210.7	7680.3	193.9
Long-n.21.0.gr	448.1	40652.3	408.7

表 26: Memory Requirement[MB]

	2-heap	buckets	mbp
Long-n.14.0.gr	1	1	2
Long-n.15.0.gr	2	2	4
Long-n.16.0.gr	4	4	7
Long-n.17.0.gr	8	8	16
Long-n.18.0.gr	17	16	30
Long-n.19.0.gr	34	32	59
Long-n.20.0.gr	67	63	118
Long-n.21.0.gr	134	126	236

C. Long-C Family に対する 1 対 1 最短路問題

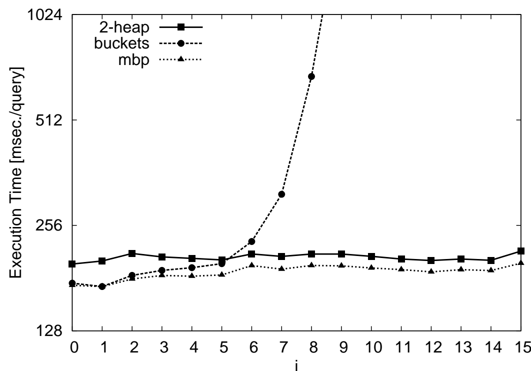


図 33: Execution Time[msec./query]

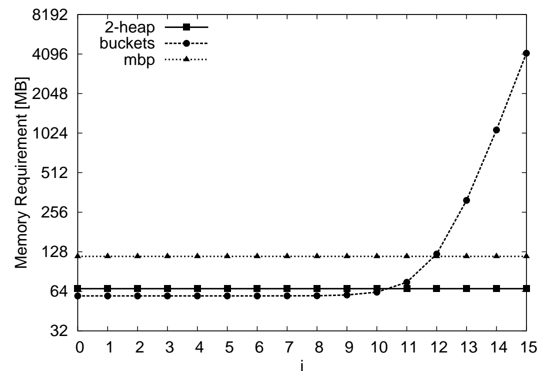


図 34: Memory Requirement[MB]

表 27: Execution Time[msec./query]

	2-heap	buckets	mbp
Long-C.0.0.gr	198.6	175.1	172.7
Long-C.1.0.gr	202.5	171.3	171.2
Long-C.2.0.gr	212.9	184.3	180.0
Long-C.3.0.gr	208.1	190.4	184.2
Long-C.4.0.gr	206.1	193.9	183.3
Long-C.5.0.gr	204.0	199.1	184.8
Long-C.6.0.gr	212.1	230.3	196.6
Long-C.7.0.gr	208.8	314.2	193.0
Long-C.8.0.gr	212.1	681.3	196.7
Long-C.9.0.gr	212.1	2076.2	196.2
Long-C.10.0.gr	208.9	7953.8	193.5
Long-C.11.0.gr	205.2	-	191.4
Long-C.12.0.gr	203.2	-	188.6
Long-C.13.0.gr	205.2	-	191.4
Long-C.14.0.gr	203.6	-	190.3
Long-C.15.0.gr	216.2	-	199.7

表 28: Memory Requirement[MB]

	2-heap	buckets	mbp
Long-C.0.0.gr	67	59	118
Long-C.1.0.gr	67	59	118
Long-C.2.0.gr	67	59	118
Long-C.3.0.gr	67	59	118
Long-C.4.0.gr	67	59	118
Long-C.5.0.gr	67	59	118
Long-C.6.0.gr	67	59	118
Long-C.7.0.gr	67	59	118
Long-C.8.0.gr	67	59	118
Long-C.9.0.gr	67	60	118
Long-C.10.0.gr	67	63	118
Long-C.11.0.gr	67	75	118
Long-C.12.0.gr	67	123	118
Long-C.13.0.gr	67	315	118
Long-C.14.0.gr	67	1083	118
Long-C.15.0.gr	67	4155	118

ⓒ日本オペレーションズ・リサーチ学会． 無断複写・複製・転載を禁ず．

D. Random4-n Family に対する 1 対 1 最短路問題

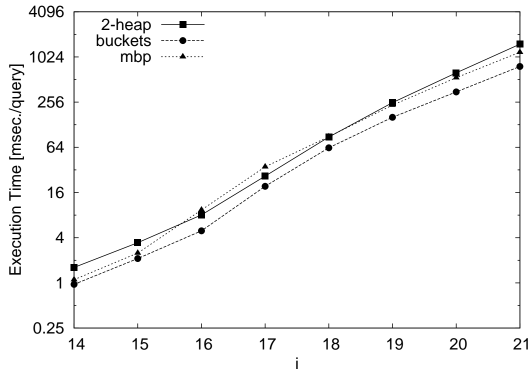


図 35: Execution Time[msec./query]

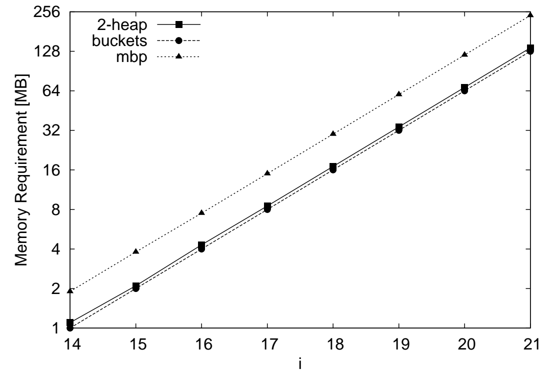


図 36: Memory Requirement[MB]

表 29: Execution Time[msec./query]

	2-heap	buckets	mbp
Random4-n.14.0.gr	1.6	1.0	1.1
Random4-n.15.0.gr	3.4	2.1	2.5
Random4-n.16.0.gr	8.0	4.9	9.4
Random4-n.17.0.gr	26.4	19.4	35.2
Random4-n.18.0.gr	87.9	63.1	88.7
Random4-n.19.0.gr	252.4	160.8	233.4
Random4-n.20.0.gr	628.0	350.3	541.3
Random4-n.21.0.gr	1516.4	765.0	1183.4

表 30: Memory Requirement[MB]

	2-heap	buckets	mbp
Random4-n.14.0.gr	1	1	2
Random4-n.15.0.gr	2	2	4
Random4-n.16.0.gr	4	4	8
Random4-n.17.0.gr	9	8	15
Random4-n.18.0.gr	17	16	30
Random4-n.19.0.gr	34	32	60
Random4-n.20.0.gr	68	64	120
Random4-n.21.0.gr	136	128	240

E. Random4-C Family に対する 1 対 1 最短路問題

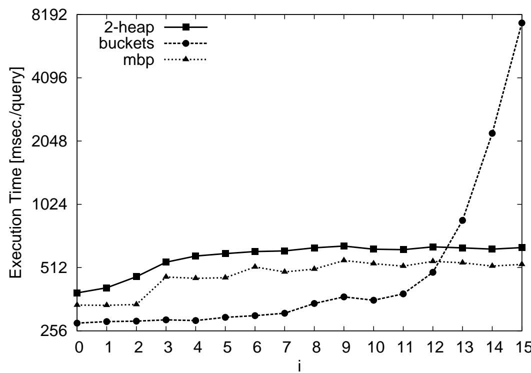


図 37: Execution Time[msec./query]

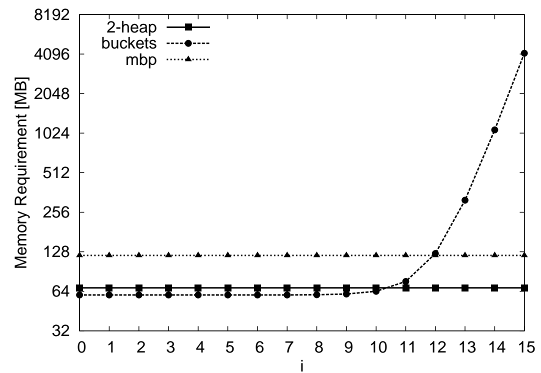


図 38: Memory Requirement[MB]

表 31: Execution Time[msec./query]

	2-heap	buckets	mbp
Random4-C.0.0.gr	387.7	278.5	339.3
Random4-C.1.0.gr	410.2	283.2	338.6
Random4-C.2.0.gr	463.0	284.7	341.7
Random4-C.3.0.gr	543.1	288.8	461.7
Random4-C.4.0.gr	581.2	286.6	455.1
Random4-C.5.0.gr	596.5	296.8	457.9
Random4-C.6.0.gr	610.2	302.4	515.4
Random4-C.7.0.gr	614.7	310.3	487.9
Random4-C.8.0.gr	636.0	345.5	503.7
Random4-C.9.0.gr	648.3	371.3	553.2
Random4-C.10.0.gr	626.1	357.8	534.6
Random4-C.11.0.gr	622.9	384.0	519.9
Random4-C.12.0.gr	642.4	486.1	548.1
Random4-C.13.0.gr	635.4	858.8	539.6
Random4-C.14.0.gr	626.6	2229.4	520.3
Random4-C.15.0.gr	637.5	7469.8	530.1

表 32: Memory Requirement[MB]

	2-heap	buckets	mbp
Random4-C.0.0.gr	68	60	120
Random4-C.1.0.gr	68	60	120
Random4-C.2.0.gr	68	60	120
Random4-C.3.0.gr	68	60	120
Random4-C.4.0.gr	68	60	120
Random4-C.5.0.gr	68	60	120
Random4-C.6.0.gr	68	60	120
Random4-C.7.0.gr	68	60	120
Random4-C.8.0.gr	68	60	120
Random4-C.9.0.gr	68	61	120
Random4-C.10.0.gr	68	64	120
Random4-C.11.0.gr	68	76	120
Random4-C.12.0.gr	68	124	120
Random4-C.13.0.gr	68	316	120
Random4-C.14.0.gr	68	1084	120
Random4-C.15.0.gr	68	4156	120

©日本オペレーションズ・リサーチ学会. 無断複写・複製・転載を禁ず.

F. Square-n Family に対する 1 対 1 最短路問題

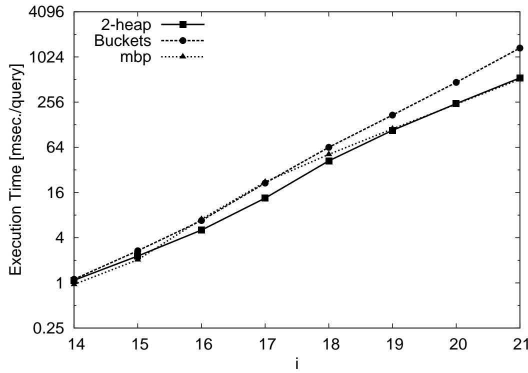


図 39: Execution Time[msec./query]

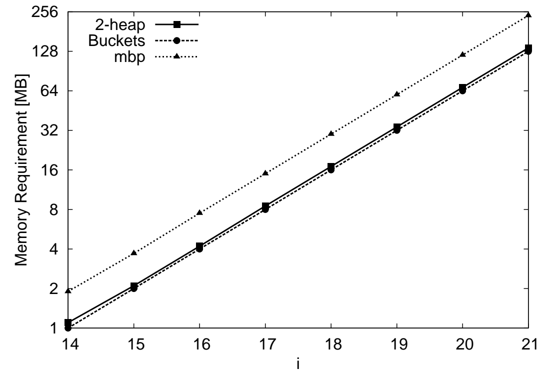


図 40: Memory Requirement[MB]

表 33: Execution Time[msec./query]

	2-heap	buckets	mbp
Square-n.14.0.gr	1.1	1.1	1.0
Square-n.15.0.gr	2.3	2.7	2.0
Square-n.16.0.gr	5.1	6.8	7.1
Square-n.17.0.gr	13.5	21.4	22.1
Square-n.18.0.gr	42.2	64.1	51.6
Square-n.19.0.gr	107.8	172.1	112.7
Square-n.20.0.gr	244.2	469.3	240.6
Square-n.21.0.gr	538.6	1343.5	517.0

表 34: Memory Requirement[MB]

	2-heap	buckets	mbp
Square-n.14.0.gr	1	1	2
Square-n.15.0.gr	2	2	4
Square-n.16.0.gr	4	4	8
Square-n.17.0.gr	9	8	15
Square-n.18.0.gr	17	16	30
Square-n.19.0.gr	34	32	60
Square-n.20.0.gr	68	64	120
Square-n.21.0.gr	136	128	240

G. Square-C Family に対する 1 対 1 最短路問題

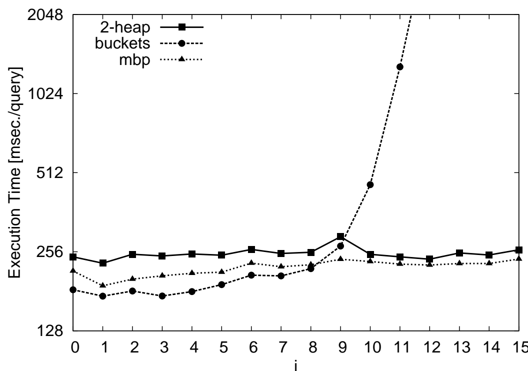


図 41: Execution Time[msec./query]

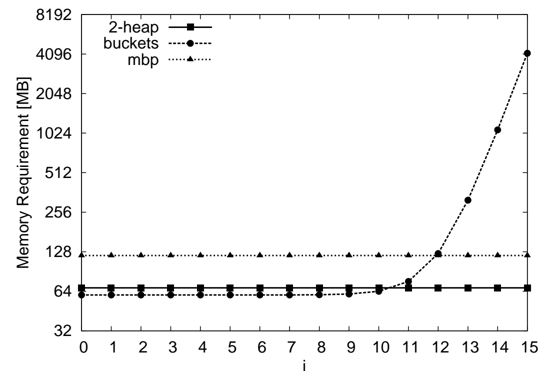


図 42: Memory Requirement[MB]

表 35: Execution Time[msec./query]

	2-heap	buckets	mbp
Square-C.0.0.gr	244.8	183.5	216.3
Square-C.1.0.gr	231.5	173.5	190.0
Square-C.2.0.gr	250.4	181.4	201.5
Square-C.3.0.gr	246.7	173.7	207.3
Square-C.4.0.gr	251.0	180.6	212.0
Square-C.5.0.gr	248.3	191.9	214.1
Square-C.6.0.gr	261.3	208.6	231.7
Square-C.7.0.gr	252.2	207.1	224.5
Square-C.8.0.gr	254.8	220.8	228.5
Square-C.9.0.gr	292.0	269.2	239.7
Square-C.10.0.gr	250.3	460.8	235.0
Square-C.11.0.gr	244.9	1293.9	229.8
Square-C.12.0.gr	239.9	4318.4	227.9
Square-C.13.0.gr	253.4	16229.0	230.9
Square-C.14.0.gr	248.9	62905.3	230.9
Square-C.15.0.gr	260.5	-	240.2

表 36: Memory Requirement[MB]

	2-heap	buckets	mbp
Square-C.0.0.gr	68	60	120
Square-C.1.0.gr	68	60	120
Square-C.2.0.gr	68	60	120
Square-C.3.0.gr	68	60	120
Square-C.4.0.gr	68	60	120
Square-C.5.0.gr	68	60	120
Square-C.6.0.gr	68	60	120
Square-C.7.0.gr	68	60	120
Square-C.8.0.gr	68	60	120
Square-C.9.0.gr	68	61	120
Square-C.10.0.gr	68	64	120
Square-C.11.0.gr	68	76	120
Square-C.12.0.gr	68	124	120
Square-C.13.0.gr	68	316	120
Square-C.14.0.gr	68	1084	120
Square-C.15.0.gr	68	4156	120

ⓒ日本オペレーションズ・リサーチ学会. 無断複写・複製・転載を禁ず.

安井雄一郎
中央大学理工学研究科経営システム工学専攻
〒112-8551 東京都文京区春日 1-13-27
E-mail: yasui@indsys.chuo-u.ac.jp

©日本オペレーションズ・リサーチ学会. 無断複写・複製・転載を禁ず.

ABSTRACT

FAST IMPLEMENTATION OF DIJKSTRA'S ALGORITHM FOR THE LARGE-SCALE SHORTEST PATH PROBLEM

Yuichiro Yasui *Chuo University*
Katsuki Fujisawa *Chuo University*
Hiroshi Sasajima *Chuo University*
Kazushige Goto *Microsoft Corporation*

The shortest path problem can be widely applied not only to route search in large-scale network but to other optimization problems where the shortest path problems are used as subproblems. Although there exist stable and efficient algorithms for the shortest path problem, we need fast implementations when solving large-scale shortest path problems. In this paper, we discuss how to make fast and general implementations of Dijkstra's algorithm, where the memory hierarchy is carefully considered to specify the bottleneck of the algorithm and to improve the performance. Our implementations with the binary heap are superior to other existing implementations when taking three factors (performance, robustness, and required computational memory) into consideration. We show that our implementations can get optimal routes very quickly and require smaller computational memory compared with other implementations through systematic numerical experiments. We also explain the Web service for large-scale shortest path problems, which employs our implementations.

©日本オペレーションズ・リサーチ学会. 無断複写・複製・転載を禁ず.