東京大学
情報理工学系研究科 創造情報学専攻
修士論文

グラフ解析プログラム向けの オブジェクトのメモリレイアウトを整列化できる Java コンパイラの開発

A Java Compiler to Optimize Memory Layout of Objects for Graph Analysis Programs

汐田 徹也
Tetsuya Shiota

指導教員　千葉 滋 教授

2014 年 1 月

概要

指定されたメソッドの呼び出し中でオブジェクトのメモリレイアウトを再整列化できるグラフ解析向け Java コンパイラを提案する．一般に，オブジェクト指向で表現されたグラフ構造では，各頂点や各辺を多様な属性を持ったオブジェクトとして抽象化するため，グラフ全体を探索しながら属性を使って計算するような処理で高いキャッシュ効率が期待できない．キャッシュ効率を高めるために，連続してアクセスするデータをメモリ上に整列させるテクニックはよく知られているが，Java 言語ではユーザがオブジェクトの整列順序による性能への影響を意識しながら開発することは難しい．そこで本研究では，プログラマの付加したアノテーションに基づくプログラム変換で，指定されたフィールドの配列化やグラフ探索順序に沿ったオブジェクト再整列化を実現する Java コンパイラを開発した．メモリレイアウトを直接制御できず，またガベージコレクションがある Java 言語で上記の機能を実現するため，ポインタオブジェクトを用いる方法を提案する．典型的なグラフ解析としてダイクストラ法を用いた実験から，この方法により最大で 10 倍以上の性能向上が見込めることを確認した．

Abstract

We propose a Java compiler that optimizes the memory layout of objects in graph analysis programs. A graph analysis program written in object orientation often shows a low cache-hit ratio if vertices and edges are objects and the program traverses vertices and accesses their attributes. Although there are several techniques of data alignment for improving a cache hit ratio, it is difficult for Java programmers to write a program while considering optimal memory layout. In this study, we developed a Java compiler that allows programmers to specify annotation for optimizing memory layout. The annotated Java program is transformed according to the annotations to implement object reordering and field arraying. To implement these functions in Java, which does not enable direct control of memory layout and provide garbage collection, we propose the implantation technique using *pointer objects*. We show performance experiments using graph analysis by the Dijkstra method and illustrate this technique improves more than 10 times speed-up in the base case.

目次

第 1 章

はじめに

一般に，オブジェクト指向で表現されたグラフ構造では，各頂点や各辺を多様な属性を持ったオブジェクトとして抽象化するため，グラフ全体を探索しながら属性を使って計算するような処理で高いキャッシュ効率が期待できない．それは，オブジェクト指向言語で記述されたプログラムではそのモジュール単位であるオブジェクト毎にメモリ上に配置されるためである．

キャッシュ効率を高めるために，連続してアクセスするデータをメモリ上に整列させるテクニックはよく知られているが，Java 言語 [1] ではユーザがオブジェクトの整列順序による性能への影響を意識しながら開発することは難しい．Java のようにオブジェクトのメモリ内配置を JVM (Java 仮想機械 [2]) に委ねるような言語では，プログラマが処理内容に応じて適切なオブジェクトのレイアウトを与える事は難しい．JVM によるオブジェクトの配置を意識して，生成順序や参照構造を注意深く調整し，間接的に実行時のメモリ内配置を制御するようなプログラミング上の工夫も可能である．しかし，それによりコードの可読性や保守性が低下するだけでなく，実行時の性能低下や，適切なガベージコレクション (GC) が実行されない問題も引き起こされる．

そこで本研究では，プログラマが付加したアノテーションに基づくプログラム変換で，指定されたフィールドの配列化やグラフ探索順序に沿ったメモリレイアウトの再整列化[*1]を実現する Java コンパイラ Javarac を開発した．ユーザはオブジェクト指向で記述されたソースコードに適切なアノテーションを付与するだけで，フィールドの配列化やオブジェクトの再整列化によって性能の向上が得られる．メモリ内レイアウトの切り替え法として，フィールド配列化とオブジェクト再整列化をサポートしている．

本論文の構成は以下の通りである．第 2 章ではオブジェクト指向言語でグラフ構造の抽象化を行う場合のメモリレイアウトの性能などの問題点を述べ，既存のメモリレイアウト最適化手法を紹介している。第 3 章では，アノテーションで指定したコード内で，オブジェクトのフィールドを配列化し再整列させられる Java コンパイラを提案する．第 4 章では第 3 章で提案したコンパイラの実装を実装した Javarac の仕様と実装上の工夫について述べ，アイディアを素朴に実装した場合と比較を行う．第 5 章では第 4 章で紹介した Javarac を用いた実験の

[*1] 動的に整列順序を変更するため本論文中では再整列と表現する

結果について説明する．第 7 章で本論文をまとめ今後の課題を示す．

第 2 章

オブジェクト指向によるグラフ構造の抽象化と関連研究

2.1　オブジェクト指向によるグラフ構造の抽象化

2.1.1　グラフ解析プログラムのオブジェクト指向による開発

グラフ解析プログラムもオブジェクト指向による設計で生産性の高い記述が可能になる．グラフの頂点や辺をオブジェクトとして表現する場合，それらに付加する属性やそれらを扱う処理はアプリケーションよって異なる．しかし，グラフ構造の探索順序は幅優先探索や深さ優先探索などの代表的な手法を拡張して扱うことが多い．例えば，グラフを純粋に深さ優先探索する場合，次に訪問する頂点は隣接する頂点で未訪問なものの中から特に制限をなく選ばれる．ここで探索順序を拡張して，「次の頂点の選択は重みが大きい辺が張られている辺の端点から貪欲的に」などのヒューリスティックを入れる場合を考える．頂点をオブジェクトとして抽象化し，頂点の選択アルゴリズムをメソッドとして記述していた場合，そのメソッドを上書きすることによって探索順序のカスタマイズが可能になる．このようにグラフ解析処理を，より一般的に用いられる部分（例では深さ優先探索の記述）とより特異な部分（例では次の訪問頂点の選択部分）に分離することにより，コードの再利用性を高めることができる．

Java で実装された代表的なグラフ解析プログラムとして，JGraphT[3] を挙げる．これは Java で実装されたグラフ解析ライブラリであり，オブジェクト指向の基づいた生産性の高い設計がされている．ソースコード 2.1 は，JGraphT を用いてユーザ定義のグラフオブジェクトに対して，ダイクストラ法 [4] を適応している例である．

大規模なグラフの解析を支援するための DSL やフレームワークがあるが，頂点や辺をオブジェクトとして扱うことができ，自由に属性を割り当てることができる．Pregel[5] は大規模グラフ解析のためのフレームワークである．GreenMarl[6] はグラフ解析向け DSL である．Pregel は用意された頂点クラスを継承して利用することができ，そこで型引数を与えることにより頂点や辺に割り当てる属性を決めることができる．GreenMarl では，コードブロックごとにあとから辺や頂点にそのコードブロックローカルな属性を割り当てることができる．さら

ソースコード 2.1: JGraphT のダイクストラ法の利用例

```
1  class MyGraph extends WeightedGraph<MyVertex, MyEdge>{
2    /* 略 */
3  }
4
5  WeightedGraph<myVertex, myEdge> g = new MyGraph();
6  MyVertex v0, v1, v2;
7  v0 = new MyVertex(0);
8  v1 = new MyVertex(1);
9  v2 = new MyVertex(2);
10 g.addVertex(v0);
11 g.addVertex(v1);
12 g.addVertex(v2);
13
14 g.setEdgeWeight(g.addEdge(v0, v1), 1);
15 g.setEdgeWeight(g.addEdge(v0, v2), 4);
16 g.setEdgeWeight(g.addEdge(v1, v2), 2);
17
18 System.out.println(DijkstraShortestPath.findPathBetween(g, v0, v2));
```

にグラフアルゴリズムの記述を支援するに様々な機能を提供する．

2.1.2　オブジェクト指向言語のメモリレイアウト

一般にオブジェクト指向言語で記述されたプログラムは，そのモジュール単位であるオブジェクト毎にメモリ上に配置される．オブジェクトの属性（フィールド）への操作を隠蔽することができるオブジェクト指向言語では，特に同一のオブジェクトのフィールドへの連続的なアクセスが起こりやすい．そのため，オブジェクトのフィールドはメモリ内にまとめて配置されると高いキャッシュヒット率が期待できる．

グラフアルゴリズムにはその過程で頂点や辺に多くの属性を割り当てる必要がある場合があり，さらにユーザ定義の属性も加わることがある．ソースコード 2.2 とソースコード 2.3 は最小費用流問題 [7] を解くプログラムをグラフの頂点や辺をオブジェクトとして抽象化して記述した例である．実装はプログラミングコンテストチャレンジブック [8] を参考にした．ソースコード 2.2 の 2 行目から 5 行目は辺の属性の定義，8 行目から 11 行目は頂点の属性を定義している．Vertex オブジェクトのメモリレイアウトの一例を図 2.1 で示す．また，メモリレイアウトを解析するツール jol [9] による Edge オブジェクトのダンプ結果 (表 2.1) が示すように，頂点や辺が持つ属性はオブジェクトごとにまとめてメモリ上に配置される．

ソースコード 2.3 の場合はオブジェクト化することにより，参照の空間局所性が向上している．Java のメモリサイズ及びオブジェクト内のメモリレイアウトは仕様 [2] で決められておらず JVM の種類によって異なるが，64bitJava プロセスでの一般的な例を図 2.1 や図 2.2 で示す．図 2.1 はソースコード 2.3 の Vertex オブジェクトに対応している．前半 192bit はオブジェクトのメタデータに使用されている．その後，フィールドに対応するメモリが連続して並んでいる．同じ頂点や辺の属性を連続して参照する例であるソースコード 2.3 では高いキャッシュヒット率が見込める．21 行目から 25 行目などは，Edge オブジェクトの属性や Vertex オ

オフセット	サイズ	型	説明
0	12	—	ヘッダ
12	4	int	Edge.cap
16	4	int	Edge.cost
20	4	Vertex	Edge.to
24	4	Edge	Edge.rev
28	4	—	パディング

表 2.1: jol による Edge オブジェクトのメモリレイアウトの取得結果の一例

ソースコード 2.2: オブジェクト指向による最小費用流問題を解くプログラムの頂点クラス及び辺クラス実装例

```
1  class Edge{
2    public Vertex to;
3    public int cap;
4    public int cost;
5    public Edge rev;
6  }
7  class Vertex{
8    public int pot, dist;
9    List<Edge> edges;
10   Edge preEdge;
11   Vertex preV;
12   void renewPot(){
13     pot += dist;
14   }
15   int getMinCapa(Vertex s){
16     if(s == this)return inf;
17     return Math.min(preEdge.cap, preV.getMinCapa(s));
18   }
19   void decreceCapacity(Vertex s, int mini){
20     if(s == this)return;
21     preEdge.cap -= mini;
22     preEdge.rev.cap += mini;
23   }
24 }
```

ブジェクトの属性に連続的にアクセスしている．対照的に配列で実装されたソースコード 2.4 は，参照が連続でない．Java における配列オブジェクトのデータレイアウトを図 2.2 で示す．また，ソースコード 2.4 の 19 行目から 24 行目までの記述は，ソースコード 2.3 の 21 行目から 25 行目などに比べて複雑になっている．

2.1.3　オブジェクトを横断するフィールドアクセス

オブジェクト指向に基づく生産性の高いデータ構造設計はグラフ処理プログラムにおいて，必ずしも高いキャッシュ効率が得られるとは限らない．グラフの頂点や辺をオブジェクトとして抽象化する場合，同一のオブジェクトに含まれるフィールドの中に，連続的にアクセスされ

ソースコード 2.3: オブジェクト指向による最小費用流問題を解くプログラムの実装例

```
1  class MinCostFlow{
2    static long minCostFlow(Graph g, Vertex s, Vertex t, int f){
3      long ret = 0;
4      for(Vertex v : g.getVertices())v.pot = 0;
5      while(f > 0){
6        /* 優先度付きキューを用いたダイクストラ法で
7           最小の重みの増加路を見つける */
8        Queue<State> Q = new PriorityQueue<State>(g.getNumOfVertex(), new
             Comparator<State>(){
9          public int compare(State s1, State s2){
10           return s1.cost - s2.cost;
11         }
12       });
13       Q.add(new State(0, s));
14       for(Vertex v : g.getVertices())v.dist = (int)1e9;
15       s.dist = 0;
16       while(!Q.isEmpty()){
17         State now = Q.poll();
18         Vertex v = now.pos;
19         if(v.dist < now.cost)continue;
20         for(Edge e : v.edges){
21           if(e.cap > 0 && e.to.dist > v.dist + e.cost +v.pot - e.to.pot){
22             e.to.dist = v.dist + e.cost + v.pot - e.to.pot;
23             e.to.preV = v;
24             e.to.preEdge = e;
25             Q.add(new State(e.to.dist, e.to));
26           }
27         }
28       }
29       if(t.dist == inf)return inf;
30       /* ポテンシャルを更新 */
31       for(Vertex v : g.getVertices()){
32         v.renewPot();
33       }
34       /* 増加路の容量を求める */
35       int mini = Math.min(f, t.getMinCapa(s));
36       f -= mini;
37       ret += mini * t.pot;
38       /* 増加路中の辺の容量を更新 */
39       v.decreceCapacity(s, mini);
40     }
41     return ret;
42   }
43 }
```

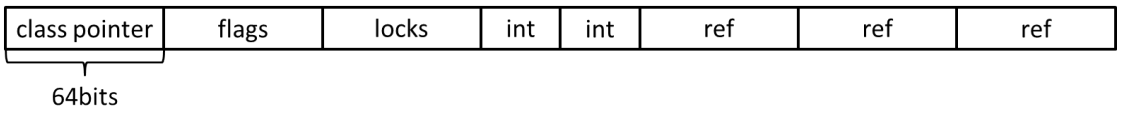

図 2.1: ソースコード 2.3 の Vertex オブジェクトのメモリレイアウトの一例

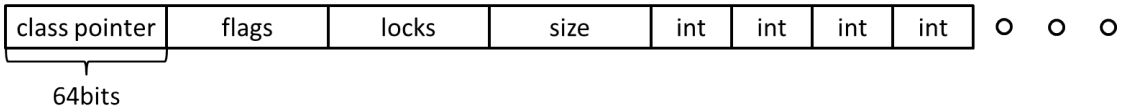

図 2.2: int[] オブジェクトのメモリレイアウトの一例

ソースコード 2.4: 配列を中心とした最小費用流問題を解くプログラムの実装例

```
1  class MinCostFlow{
2    static long minCostFlow(Graph g, int s, int t, int f){
3      long ret = 0;
4      for(int i = 0; i< g.pot.length; i++)g.pot[i] = 0;
5      while(f > 0){
6        Queue<State> Q = new PriorityQueue<State>(g.getNumOfVertex(), new
             Comparator<State>(){
7          public int compare(State s1, State s2){
8            return s1.cost - s2.cost;
9          }
10       });
11       Q.add(new State(0, s));
12       for(int i = 0; i<g.getNumOfVertex(); i++)g.dist[i] = (int)1e9;
13       g.dist[s] = 0;
14       while(!Q.isEmpty()){
15         State now = Q.poll();
16         int v = now.pos;
17         if(g.dist[v] < now.cost)continue;
18         for(int e = 0; e<g.to[v].length; e++){
19           if(g.cap[v][e] > 0 && g.dist[g.to[v][e]]> g.dist[v] + g.cost[v][e]+
                 g.pot[v] - g.pot[g.to[v][e]]){
20             g.dist[g.to[v][e]] = g.dist[v] + g.cost[v][e] + g.pot[v] -
                   g.pot[g.to[v][e]];
21             g.preV[g.to[v][e]] = v;
22             g.preE[g.to[v][e]] = e;
23             Q.add(new State(g.dist[g.to[v][e]], g.to[v][e]));
24           }
25         }
26       }
27       if(g.dist[t] == 1e9)return (int)1e9;
28       for(int i = 0; i<g.pot.length; i++){
29         g.pot[i] += g.dist[i];
30       }
31       int mini = f;
32       for(int v = t; v != s; v = g.preV[v]){
33         mini = Math.min(mini, g.cap[g.preV[v]][g.preE[v]]);
34       }
35       f -= mini;
36       ret += mini * g.pot[t];
37       for(int v = t; v != s; v = g.preV[v]){
38         g.cap[g.preV[v]][g.preE[v]] -= mini;
39         g.cap[v][g.rev[g.preV[v]][g.preE[v]]] += mini;
40       }
41     }
42     return ret;
43   }
44 }
```

るフィールドとされないフィールドが存在することがある．その場合，他のフィールドに比べアクセス頻度の低いフィールドや共に使われることのないフィールド同士が，メモリ内に隣接してしまうような非効率なレイアウトが構築されてしまう．ソースコード 2.5 に示すダイクストラ法の実装例では，Vertex オブジェクトのフィールドとして始点からの距離 (dist) や接合する辺集合 (edges)，訪問済みかを保持するフラグ (visited) が定義され，Edge オブジェクトのフィールドとしては辺の端点 (dst) や重み (weight) などが定義されている．このようなダイクストラ法の実装では，未訪問な頂点の中で距離が最小である頂点をグラフ全体から探すような処理を含むプログラム (dijkstra() メソッド) で，頂点オブジェクトにまたがる visited フィール

ソースコード 2.5: オブジェクト指向によるダイクストラ法の実装例

```
1  class Vertex{
2    public boolean visited;
3    public int dist;
4    private Edge[] edges;
5    public Edge[] getEdges(){
6      return edges;
7    }
8  }
9
10 public void dijkstra(Vertex s, Graph g){
11   for(Vertex v : g.getVertexList()) v.dist = inf;
12   s.dist = 0;
13   while(true){
14     Vertex miniV = null;
15     for(Vertex v : g.getVertexList()){
16       if(v.visited)continue;
17       if(miniV == null || miniV.dist > v.dist){
18         miniV = v;
19       }
20     }
21     if(miniV == null)break;
22     miniV.visited = true;
23     for(Edge e : miniV.getEdges()){
24       e.dst.dist = min(miniV.dist + edge.weight, e.dst.dist);
25     }
26   }
27 }
```

ドの連続的なアクセスが必要になってくる．visited フィールドがメモリ内に隣接してまとめられていない場合，キャッシュ効率が著しく低下してしまう．さらに最短経路問題では，頂点オブジェクトに経路復元のために直近に訪問した頂点を保持するフィールドや，ポテンシャルを扱うためのフィールドなど複数のフィールドが定義されることが考えられる．その結果，頂点オブジェクトのサイズが増大し，キャッシュ効率が低下すると，オブジェクト指向に基づくデータ構造による性能劣化が無視できなくなる．

しかし，Java のようにオブジェクトのメモリ内配置を JVM (Java 仮想機械) に委ねるような言語では，プログラマが処理内容に応じて適切なオブジェクトのレイアウトを与える事は難しい．オブジェクトを横断して特定のフィールドを連続的にアクセスする場合でのキャッシュ効率を向上させるオブジェクト設計には，間接的にメモリ内レイアウトを調節するテクニックとしてフィールドを配列化する方法が知られている．Strzodka [10] は C 言語において，マクロを使用して，構造体の配列 (AoS: Array of Structs) と配列の構造体 (SoA: Struct of Arrays) の切り替えを実現している．構造体データの配列が，そのメンバ毎の配列に変換されるため，連続した構造体のメンバがメモリ内に隣接して配置されるようになる．ソースコード 2.5 について Strzodka の手法を参考に，visited フィールドを配列化した例をソースコード 2.6 に示す．visited フィールドは配列化され，Vertex オブジェクトの static なフィールドとして定義されるため，オブジェクトをまたがる連続アクセスの性能向上が見込める．しかし，配列化した visited フィールドや，それに伴う辺や他の頂点オブジェクトへのアクセスは，添字を用いた配列アクセスとなる．そのため，全てのオブジェクトがアクセス可能となり，カプセ

ソースコード 2.6: フィールド配列化の実装例

```
1  class Vertex{
2    public static boolean[] visitedList;
3    public static Vertex[] objList;
4    public int dist;
5    private Edge[] edges;
6    public Edge[] getEdges(){
7      return edges;
8    }
9  }
10
11 public void dijkstra(int s, Graph g){
12   // 非配列化フィールドへのアクセス
13   for(int v : g.getVertexList()) {
14     Vertex.objList[v].dist = inf;
15     Vertex.visitedList[v] = false;
16   }
17   Vertex.objList[s].dist = 0;
18   while(true){
19     int miniV = -1;
20     for(int v : g.getVertexList()){
21       // 配列化フィールドへのアクセス
22       if(Vertex.visitedList[v])continue;
23       if(miniV == -1 || Vertex.objList[miniV].dist > Vertex.objList[v].dist){
24         miniV = v;
25       }
26     }
27     if(miniV == -1)break;
28     Vertex.visitedList[miniV] = true;
29     // 略
30   }
31 }
```

ル化によるデータ隠蔽などのオブジェクト指向の恩恵が犠牲となってしまう．

また，配列化したフィールドへの連続アクセスで，必ずしも高いキャッシュ効率を得られるとは限らない．もし配列へのアクセス順序がフィールドの整列順序に沿わないランダムアクセスであれば，キャッシュサイズを超えるアクセスによってキャッシュミスが多発してしまう．一方，たとえ配列へのアクセス順序が整列順序に沿っていたとしても，元のオブジェクトがアクセス順序に対してランダムに並んでいるとキャッシュ効率は良くならない．配列化したフィールドへのランダムアクセスは，配列データを並び替えて整列し，それに沿って各オブジェクトの保持する添字を振り直せば回避することもできる．ところが，整列したフィールドの順序に元のオブジェクトを並び替えることは困難である．特に，Java ではオブジェクトの整列順序をプログラマが陽に与えることができない．典型的な JVM では，オブジェクト生成時のメモリ配置が一定期間保たれるため，配列の初期化時にのみ間接的にオブジェクトを整列させられる可能性はある．しかし，あくまでオブジェクトの整列順序が保たれるのは，GC による断片化したメモリ領域のコンパクションが発生するまでに限られる．最適化された GC のコンパクションフェーズでは，不要オブジェクトを解放したメモリの空き領域を詰めるだけでなく，オブジェクトをその参照関係やアクセス頻度に応じて再配置する．したがって，単に特定のフィールドのみを配列化しただけで，そのアクセスが常に高い局所性をもたらす訳ではない．

2.2　既存のデータレイアウト最適化手法

従来，実行時のアクセス傾向に応じて，自動的にオブジェクトの構造を切り替える手法は数多く提案されてきた．Chilimbi らは，オブジェクトを分割してキャッシュヒット率の向上を図る structure splitting[11] を提案している．この手法では，実行前のプロファイリングにより，オブジェクト内のフィールドををアクセス頻度の高いフィールドと低いフィールドに分類する．そして，元のオブジェクトにはアクセス頻度の高いフィールドを残し，アクセス頻度の低いフィールドは別のオブジェクトとして分離し，元のオブジェクトに参照を持たせる．分割後にはアクセス頻度の高いフィールドが局所化され，参照の局所性が向上しキャッシュヒット率が向上することが期待できる．structure splitting は Java に適用され，プロファイラから分割アルゴリズム，コード変換までが実装され自動化されている．前田らの研究 [12] は Chilimbi らの手法を応用し，実行時にオブジェクトレイアウトを切り替え，共有メモリ環境でのキャッシュミスを抑えている．具体的には，オブジェクトのフィールドを書き込み頻度や読み込み頻度から 3 種類に分類し，並列プログラミングで問題になる変数の一貫性の破壊によるキャッシュミスに配慮したメモリレイアウトを実現している．さらに，事前のプロファイリングによるレイアウトの決定では動的にアクセスパターンが変化する場合への対応が難しいことを指摘し，実行時にアクセスパターンを解析しレイアウトを切り替える手法を提案している．また，Franz らも頻度によってオブジェクトを分割する手法を提案している．[13]

Structure Splitting の例をソースコード 2.7，ソースコード 2.8，図 2.3 で示す．ソースコード 2.7 のようなアクセス頻度に偏りがあるクラス定義があるとする．フィールド fld0, fld1 のみ頻繁にアクセスされるとする．このクラスのオブジェクトのメモリ上のレイアウトを図 2.3 の A で示す．高頻度にアクセスされるフィールドと低頻度にアクセスされるフィールドが混在している．Structure Splitting を行った例をソースコード 2.8 と図 2.3 の B で示す．アクセス頻度が高い fld0，fld1 をまとめて配置することに成功し，キャッシュヒット率が向上することが見込める．

フィールドの配置順序も性能に影響を与える．Chilimbi らは [11] の研究で同時に Field Reordering という手法を提案し，C/C++ 向けにキャッシュを考慮したフィールドのリコメンドを行うツールを作成している．また，Truong らは Field reorganization という名称でフィールドの配置順序を最適化している [14]．さらに, Turong らは同時に, StructureSplitting のような効果を得ることができる，Instance interleaving を提案している．

キャッシュ効率を高める手法ではないが，Perry らの研究 [15] では，構造体などの分割を MPI[16] の転送速度の向上のために用いている例を紹介する．これは，ある構造体において転送したいフィールドと転送を避けたいフィールドが存在する場合，転送が必要なフィールドのみ分割する方法である．送受信する構造体と，実際にプログラム中で用いたい構造体の構成要素が食い違った場合，転送後コピーする必要が生じる．送信側の転送に用いる構造体へのパッキングと受信側のアンパッキングのコストを抑えることが目的である．

GC を行う過程でデータのコピーが必要になるが，そのコピーを行う際にオブジェクトの整

ソースコード 2.7: Structure Splitting が有効な例

```
1  class A{
2      /* アクセス頻度高 */
3      Type0 fld0;
4      Type1 fld1;
5
6      /* アクセス頻度低 */
7      Type2 fld2;
8      Type3 fld3;
9      Type4 fld4;
10 }
```

ソースコード 2.8: Structure Splitting を行った例

```
1  class A{
2      Type0 fld0;
3      Type1 fld1;
4      cld_A cld;
5  }
6
7  class cld_A{
8      Type2 fld2;
9      Type3 fld3;
10     Type4 fld4;
11 }
```

列化を行う研究もよく知られている．Cheney らは，参照の関係を表したグラフを幅優先探索順にコピーし [17]，Moon らは深さ優先探索探索順にコピーしている [18]．Chilimbi らは実行時にアクセスパターンのプロファイリングを行った結果から，キャッシュ効率を考慮したコピーアルゴリズムを提案している [19]．具体的には，プロファイリングからオブジェクトの関係性を重み付きグラフで表現し，そのグラフ上で深さ優先探索を行いその順序でオブジェクトのコピーを行う．その際，あるノードから複数の辺が伸びている場合，端点が未訪問なオブジェクトでかつ最も重みが大きい辺を貪欲的に選択する．その結果，プログラムの実行性能が Cheney らの手法や，オブジェクトの構造や型などの情報を用いたヒューリスティックを採用している手法 [20] より大きく高速化している．Huang らは参照頻度の高いオブジェクトと低いオブジェクトに分類し，参照頻度の高いオブジェクトのための領域を設置しキャッシュヒット率の向上を図っている [21]．それぞれのクラスについてフィールドについて，hot field であるかどうかを管理し，hot field をもつクラスのオブジェクトや hot field から参照されているオブジェクトなどを hot object と分類している．hot object を特定のメモリ領域に集めることで，アクセスされやすいオブジェクトを局所化しパフォーマンスの向上を図っている．一方，Novark らは JVM を改変し，GC 時のオブジェクトの配置を開発者が柔軟にカスタマイズできるようにしている [22]．開発者は，システムが提供するインターフェース CustomLayout を実装し，GC をトリガーとしてどのような順でオブジェクトを並び替えるかのアルゴリズムを実装できる．三種類の木構造に対して実験を行い，構造に合わせた整列による性能の向上を確認している．

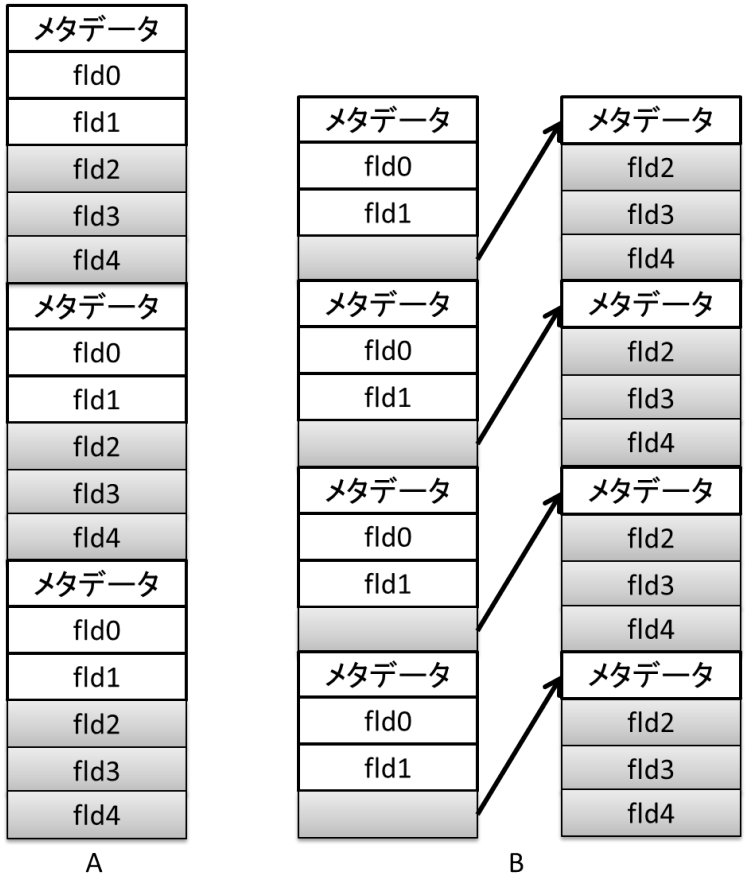

図 2.3: ソースコード 2.7 に対応するメモリレイアウト (A) とソースコード 2.8 に対応するメモリレイアウト (B)

メモリを直接操作できる C/C++ では，Java に比べて，比較的容易にデータ配置の切り替えによる最適化が可能である [23][24]．Wang らは実行時にバイナリレベルでの structure splitting[25] による最適化を提案している．Strzodka は OpenCL[26] を利用したプログラムを対象に，マクロを使用して構造体の配列 (AoS: Array of Structs) と配列の構造体 (SoA: Struct of Arrays) の切り替えを実現している [10]．

近年，盛んに研究が進んできた大規模グラフ解析を支援するための専用言語やフレームワークでも，頂点や辺をオブジェクトとして扱うことができ，自由に属性を割り当てることができる．Pregel [5] は，Bulk Synchronous Parallel を実装し，反復的な頂点処理を簡素化できる大規模並列グラフ解析のためのフレームワークである． 開発者は頂点クラスを継承し型引数を与えることにより，頂点や辺に割り当てる様々な属性を設定できる．一方，グラフ解析向けのソースコードの高い抽象化を目指した専用言語 GreenMarl[6] では，並列処理や探索順序の切り替えを簡潔に行え，頂点や辺が持つ一時的な属性を配列として追加する機能が提供されている．本研究は，このようなグラフ解析向けのフレームワークや専用言語を開発する上で，その効率的な実装にも寄与できると考えられる．

第 3 章

フィールドを配列化しオブジェクト再整列させられる Java コンパイラ

本研究ではアノテーションで指定したコード内で，オブジェクトのフィールドを配列化し再整列できる Java コンパイラを提案する．ユーザはオブジェクト指向で記述されたソースコードに適切なアノテーションを付与するだけで，フィールドの配列化やオブジェクトの再整列化による性能の向上が得られる．メモリ内レイアウトの切り替え法として，フィールド配列化 *(Field Arraying)* とオブジェクト再整列化 *(Object Reordering)* をサポートしている．使用例をソースコード 3.1 に示す．

配列化するフィールドには，Arraying/Reserved アノテーションを用い，そのフィールドの定義クラスには Target アノテーションを付与する．ソースコード 3.1 の Vertex クラスの例のように，Target アノテーションが付与された Vertex クラスのオブジェクトでは，Arraying/Reserved アノテーションが付与されたフィールドのみ配列化される．Reserved アノテーションが付与されたフィールド visited と dist は配列化され，Vertex オブジェクトをまたがった参照がコンパイラによって連続した配列アクセスへ変換される．したがって，ユーザは 23–25 行目のように，配列化されたフィールドを意識する事なく，Vertex クラスのフィールドとしてアクセスできる．

Reserved アノテーションの付与されたフィールドを格納する配列は動的に確保され，AllocateFields アノテーションの付与されたメソッド内でのみ有効となる．オブジェクトをまたがってアクセスされるフィールドは，グラフを巡回する一時的な計算に利用される場合が多い．そのため，19 行目のように AllocateFields アノテーションの付与された dijkstra メソッド呼び出しされる直前にメモリ領域を確保し，呼び出し終了時に解放すれば，無駄なメモリを確保し続けることを避けられる．一方，Arraying アノテーションが付与されたフィールドの配列は事前に確保され，AlloateFields アノテーションが付与された箇所以外でもアクセスし続けられる．AllocateFields アノテーションのパラメタには，有効化したいフィールドの情報を¡クラス名¿.¡フィールド名¿の形式で文字列の配列として与える．

オブジェクトの再整列化は，Reorder アノテーションによって指定する．整列化を行う順序は 18 行目のように，Reorder アノテーションの引数に対象オブジェクトを目的の並び順で並

ソースコード 3.1: アノテーションによるメモリ内レイアウト変更例

```
1  @Target
2  class Vertex{
3    @Reserved
4    public boolean visited;
5    public int dist;
6    private List<Edge> edges;
7    Vertex(){
8      edges = new ArrayList<Edge>();
9    }
10   public List<Edge> getEdges(){
11     return edges;
12   }
13   public void addEdge(Edge e){
14     edges.add(e);
15   }
16 }
17
18 @Reorder(classes = {"Vertex"}, orders = {"g.getAllVertices()"})
19 @AllocateFields({"Vertex.visited"})
20 static int dijkstra(Graph g, Vertex src, Vertex dst){
21     List<Vertex> vertices = g.getAllVertices();
22     for(Vertex v :vertices){
23       v.dist = inf;
24       v.visited = false;
25     }
26     src.dist = 0;
27     while(true){
28       Vertex miniV = null;
29       for(Vertex v : vertices){
30         if(v.visited)continue;
31         if(miniV == null || (miniV.dist > v.dist)){
32           miniV = v;
33         }
34       }
35       if(miniV == null)break;
36       miniV.visited = true;
37       for(Edge e : miniV.getEdges()){
38         e.dst.dist = Math.min(e.dst.dist, miniV.dist + e.weight);
39       }
40     }
41     return dst.dist;
42   }
43 }
```

べたリストを返す式として与える．元のオブジェクトの並びは，各オブジェクトを再生成することによってリスト中の Vertex オブジェクトの整列順序に倣って並び替えられる．配列化したフィールドのデータはリストの順にならって配列内で並び替えられる．もし，アノテーションの引数の式で返すリストが，対象となるクラスのオブジェクトを全て含んでいない場合，与えられた順序に倣って整列した後，残りのオブジェクトが元の順序に従って並び追加される．対応する配列化したフィールドも同様に整列化される．

Reorder アノテーションが付与されたメソッドでは，呼び出し時にユーザが指定した順序にオブジェクトが整列され，同時にガベージコレクションが行われる．並び替えの概要を図 3.2 で示す．また，付加されたアノテーションによってメソッド呼び出しがどのように変化するかを図 3.1 で示す．整列は対象となるオブジェクトのリストをユーザが与え，その順序に倣って

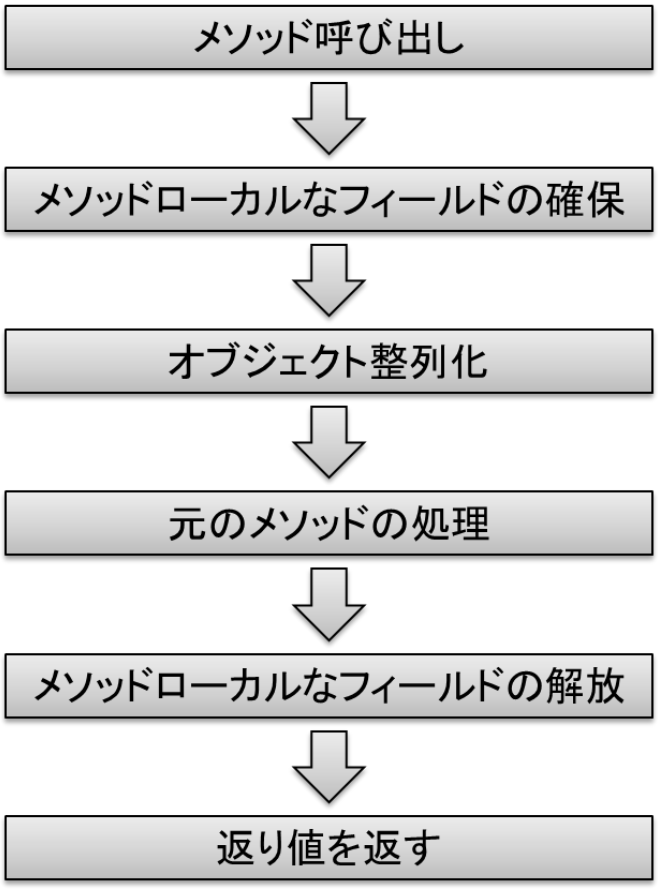

図 3.1: メソッド呼び出しの変換概要

オブジェクトを再生成することによって行われる．与えたリストが対象となるクラスのオブジェクトを全て含んでいない場合，与えられた順序に倣ってオブジェクトを再生成した後，残りのオブジェクトが元の順序に従って再生成される．

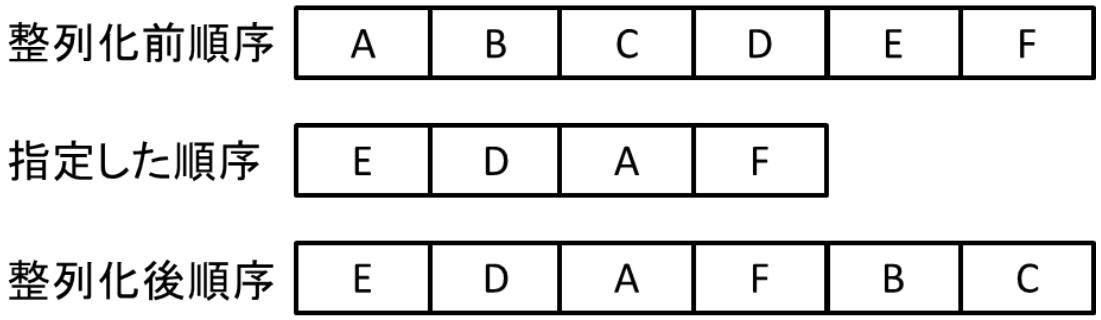

図 3.2: 指定した順序とオブジェクト再整列の関係

第 4 章

コンパイラによる GC と性能を考慮したフィールドの配列化とオブジェクト再整列化

我々は前章で述べたフィールド配列化とオブジェクト再整列化を実現する拡張 Java コンパイラ *Javarac: Java Reordering and Arraying Compiler* を開発した．Javarac はユーザが付加したアノテーションにしたがって，静的にクラス定義やフィールドへのアクセス方法を変更し，メモリレイアウト変更のためのコードを挿入する．Javarac は，コンパイラ拡張フレームワーク JastAdd[27] で実装された Java コンパイラ JastAddJ[28] を拡張し，ソースコードに付与されたアノテーションに基づいて抽象構文木レベルでコード変換を行う．コンパイラはコンパイラ拡張フレームワーク JastAdd[27] で実装された Java コンパイラ JastAddJ[28] を拡張し実装した．コンパイラはソースコード 3.1 のようなコードを受け取ると，ソースコード 4.4 やソースコード 4.1，ソースコード 4.2 のようなコードを生成する．クラス定義の変更をソースコード 4.4 で示し，アクセス部の変更やオブジェクト再整列や Reserved されたフィールドの有効化のためのコード挿入の例などをソースコード 4.1 で示す．

4.1 Pointer オブジェクトの導入

コンパイラはフィールド配列化や整列化を行う対象のオブジェクトの参照をすべて Pointer オブジェクトに置き換える．Pointer オブジェクトの定義をソースコード 4.1 で示す．Poiner オブジェクトは元のオブジェクトは配列化されたフィールドへのアクセスを中継する．元のオブジェクトにそれぞれ固有の添字が振られ，Poitner オブジェクトは添字を保持し元のどのオブジェクトに対応しているかを保持する．Pointer オブジェクトの導入に伴うコード変換は主に以下の通りである．

- クラス定義の変換
- 変数や引数，型引数などの型の変換

ソースコード 4.1: コンパイラが生成する dijkstra メソッドのコード

```
1  static int dijkstra$original(Graph g, objectrotating.Pointer src,
       objectrotating.Pointer dst) {
2    List<objectrotating.Pointer> vertices = g.getAllVertices();
3    for (objectrotating.Pointer v : vertices) {
4      ((normal.Vertex)normal.Vertex.data.objList.get(v.id)).dist = inf;
5      normal.Vertex.visited[v.id] = false;
6    }
7    ((normal.Vertex)normal.Vertex.data.objList.get(src.id)).dist = 0;
8    while(true){
9      objectrotating.Pointer miniV = null;
10     for (objectrotating.Pointer v : vertices) {
11       if(normal.Vertex.visited[v.id])
12         continue ;
13       if(miniV == null ||
             (((normal.Vertex)normal.Vertex.data.objList.get(miniV.id)).dist >
             ((normal.Vertex)normal.Vertex.data.objList.get(v.id)).dist)) {
14         miniV = v;
15       }
16     }
17     if(miniV == null)
18       break ;
19     normal.Vertex.visited[miniV.id] = true;
20     for (Edge e :
           ((normal.Vertex)normal.Vertex.data.objList.get(miniV.id)).getEdges()) {
21       ((normal.Vertex)normal.Vertex.data.objList.get(e.dst.id)).dist =
             Math.min(((normal.Vertex)normal.Vertex.data.objList.get(e.dst.id)).dist,
             ((normal.Vertex)normal.Vertex.data.objList.get(miniV.id)).dist +
             e.weight);
22     }
23   }
24   return ((normal.Vertex)normal.Vertex.data.objList.get(dst.id)).dist;
25 }
```

ソースコード 4.2: コンパイラが生成する挿入する dijkstra メソッドの前後処理のコード

```
1  static int dijkstra(Graph g, objectrotating.Pointer src, objectrotating.Pointer
       dst) {
2    int[] Vertex_mapping_ = Rotatable.reorder(g.getAllVertices(), Vertex.data);
3    Vertex.fieldReorder(Vertex_mapping_);
4    Vertex.visited = Rotatable.allocateArray(Vertex.visited, Vertex.data);
5    int _ret_ = dijkstra$original(g, src, dst);
6    Vertex.visited = null;
7    return _ret_;
8  }
```

ソースコード 4.3: コンストラクタの変換例

```
1 // T: Target アノテーションが付与されたクラス
2 // t: T 型の変数
3 t = new T(); // 変換前
4 t = new T().getNewInstance(); // 変換後
```

ソースコード 4.4: クラス定義変換の例

```
1  public class Vertex extends objectrotating.Rotatable {
2    public int dist;
3    // 配列化されたフィールド
4    public static boolean[] visited;
5    private List<Edge> edges;
6    Vertex() {
7      super();
8      edges = new ArrayList<Edge>();
9    }
10   public List<Edge> getEdges() {
11     return edges;
12   }
13   // 配列化した再整列用のメソッド
14   public static void fieldReorder(int[] order) {
15     visited = reorderFields(order, visited);
16   }
17   public static objectrotating.Util data;
18   static {
19     data = new objectrotating.Util();
20   }
21 }
```

ソースコード 4.5: Pointer クラスの定義

```
1  public class Pointer implements Cloneable{
2    public int id;
3    public Pointer(){
4    }
5    public Pointer(int id){
6      this.id = id;
7    }
8    public Pointer clone(){
9      /* 略 */
10   }
11 }
```

- コンストラクタの変換

コンパイラは Target アノテーションが付与されたクラスの定義を書き換える．クラス定義の変換は，配列化対象のフィールドの配列化や Pointer オブジェクト導入のためのメソッドを実装した抽象クラスの継承などが行われる．ソースコード 4.4 ではクラス定義の変換の例を示している．コンパイラは Target アノテーションが付与されたクラスの型の変数宣言を変換する．

また，変数宣言は型をすべて Pointer 型に変換し，さらにコンストラクタでは作られた元のオブジェクトに対応する Pointer 型を返すように変換する．変換の例をソースコード 4.6 で示す．getNewInstance はレシーバに対応する Pointer オブジェクトを返すメソッドである．クラス定義変換時に Reorderable クラスを継承させることによって利用可能になっている．Reorderable の定義を 4.10 で示す．

ソースコード 4.6: コンストラクタの変換例

```
1 // T: Target アノテーションが付与されたクラス
2 // t: T 型の変数
3 t = new T(); // 変換前
4 t = new T().getNewInstance(); // 変換後
```

ソースコード 4.7: 変換の概要

```
1  // T: Target アノテーションが付与されたクラス
2  // t: T 型の変数
3  // f1: 配列化対象でないクラス T フィールド
4  // f2: 配列化対象のクラス T フィールド
5  // m: クラス T のフィールド
6
7  // 変換前
8  t.f1;
9  t.f2;
10 t.m();
11
12 // 変換後
13 T.data.objList.get(t.id).f1;
14 T.f2[t.id];
15 T.data.objList.get(t.id).m();
```

もとのオブジェクトへのアクセスや配列化したフィールドへのアクセスは Pointer オブジェクトを介したものに変換する．変換の概要をソースコード 4.7 で示す．アクセスの変換は以下の三種類である．

- 配列化対象でないフィールドのアクセス (ソースコード 4.7 の 8 行目)
- 配列化対象であるフィールドへのアクセス (ソースコード 4.7 の 9 行目)
- メソッド呼び出し (ソースコード 4.7 の 10 行目)

コンパイラによりソースコード 4.7 の 13 行目から 15 行目のように変換される．ソースコード 4.7 の T は図 4.2 中の対象クラスに対応し，T.data はクラス T の静的領域の変数である．T.data は Util オブジェクトへの参照である．objList は図 4.2 中の対象オブジェクトの参照リストへの参照である．また，T.f2 は配列化されたフィールドを格納する配列の参照を持つ．

Pointer オブジェクトや元のオブジェクトの参照，配列化したフィールドはクラスごとに管理される．変換されたデータや Poitner オブジェクトの参照関係を図 4.2 で示す．また，Pointer オブジェクトを利用した参照方法などの概念図を図 4.1 で示す．変換対象のクラスの静的領域から，Poitner オブジェクトや元のオブジェクト，配列化したフィールドは参照されている．Util オブジェクトは変換対象のクラスそれぞれに一つ所持される．そのため，変換対象のクラスが複数存在しても，区別して扱うことが可能である．Util オブジェクトの定義を 4.8 で示す．Util オブジェクトは元のオブジェクト全てを弱参照で管理する．

ソースコード 4.1 の例では，3 行目は通常のフィールドへのアクセス，4 行目は配列化したフィールドへのアクセスを行っている．配列化したフィールド (visited) は対象クラス (Vertex)

ソースコード 4.8: Util クラスの定義

```
1 public class Util {
2   public List<List<WeakReference<objectrotating.Pointer>>> pointers;
3   public List<Rotatable> objList;
4   public Util() {
5     pointers = new ArrayList<List<WeakReference<objectrotating.Pointer>>>();
6     objList = new ArrayList<Rotatable>();
7   }
8 }
```

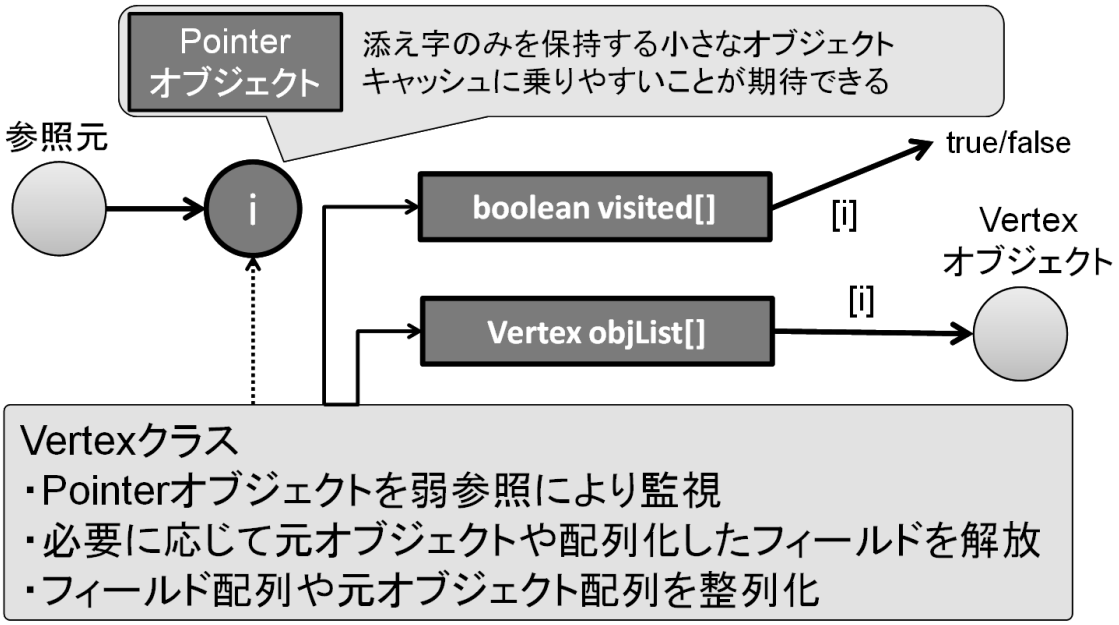

図 4.1: ポインタオブジェクトを介したオブジェクトアクセス

のクラスフィールドとして管理される．元のオブジェクト (objList) は対象クラス (Vertex) ごとに Util オブジェクト (Vertex.data) が管理する．9 行目はメソッド呼び出しを行っている．通常のフィールドへのアクセスと同様に，もとのオブジェクトにアクセスしている．

4.2　ポインタオブジェクトによって解決された問題

4.2.1　性能

Pointer オブジェクトへのアクセスは元のオブジェクトと比較して高速である．Pointer オブジェクトは元のオブジェクトの添字のみをフィールドを持つ小さなオブジェクトである．このため，元のオブジェクトと比較して Pointer オブジェクトへの参照は高いキャッシュヒット率が期待できる．

Pointer オブジェクトを介した参照は，間接参照が増加し元のオブジェクトへのアクセスは性能が悪化するが，配列化したフィールドへのアクセスは高速で行える．Pointer オブジェク

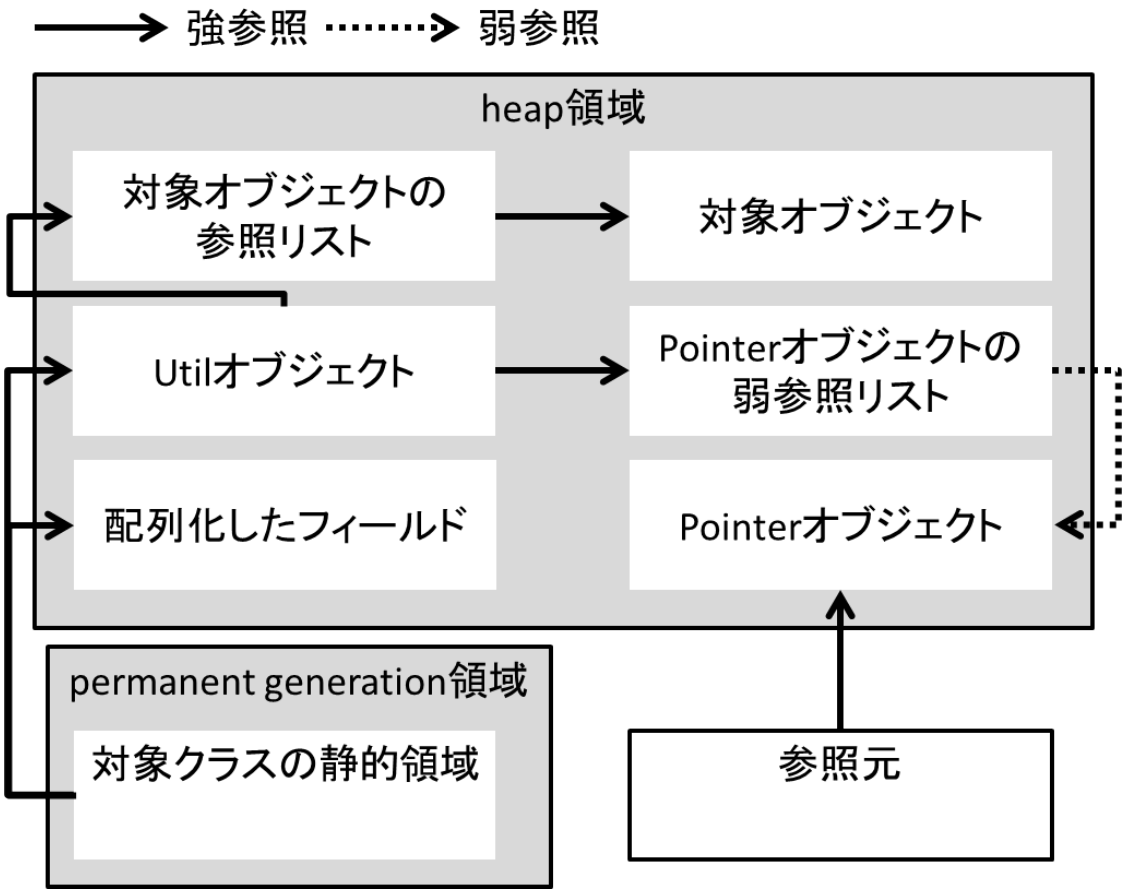

図 4.2: 変換後の参照関係

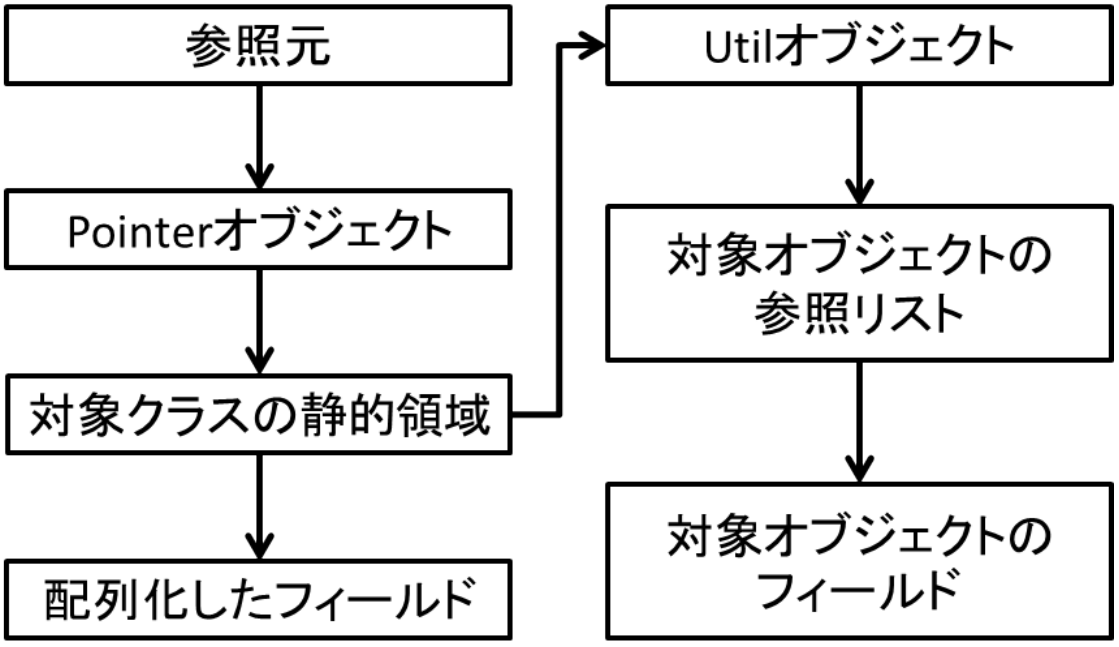

図 4.3: Pointer オブジェクトを利用したフィールドアクセス

トを介した参照では，元のオブジェクトへのアクセスは間接参照が増えるだけであり，性能が悪化する．一方，配列化されたフィールドへのアクセスは元のオブジェクトへのアクセスを伴わず，Ponter オブジェクトと配列へのアクセスのみであるので，オーバーヘッドが小さい．

4.2.2　ガベージコレクション

Pointer オブジェクトの生死を管理することにより，不要になったオブジェクトを検出する．Pointer オブジェクトは元のオブジェクトの参照の代わりに導入されているので，変換前のプログラムで元のオブジェクトがガベージコレクタに回収されるタイミングで，Pointer オブジェクトが回収される．Pointer オブジェクトすべてのリストを弱参照で持つことによって，Pointer オブジェクトの回収を確認することができる．

Pointer オブジェクトの回収を検出し，元のオブジェクトの参照を切り，添字を振りなおすことによって元のオブジェクトや配列化されたフィールドのメモリを解放する．Pointer オブジェクトの回収を検知すると，保持している元のオブジェクトのリストから対応するオブジェクトへの参照を切る．これにより，元のオブジェクトはガベージコレクタに回収される．しかし，不要なオブジェクトに対応するフィールドを格納する配列のメモリは解放されない．これは，メモリが解放されないだけでなく，データが連続化されずキャッシュヒット率の低下を引き起こす．そのため，コンパイラはオブジェクトに対応する添字を振り直すコードを挿入する．これにより，適切な長さの配列に変更できメモリを解放することができ，さらに，有効なデータが連続化され性能の向上が期待できる．これらの処理はオブジェクトの再整列と同時に行われる．

4.2.3　オブジェクトの再整列

全てのオブジェクトへの参照を Pointer オブジェクトを介し間接参照を増やすことによって，オブジェクトの再整列を可能にする．元のオブジェクトには添字を用いてアクセスするので，オブジェクトを再生成しても元のオブジェクトを管理するリストの参照を再生成先のものに書き換えれば，すべてのアクセスが再生成したオブジェクトへのアクセスに変換される．

添字を振りなおす場合は Pointer オブジェクトの添字を全て更新することによって実現可能である．添字を振りなおすことによって，配列化されたフィールドへのアクセスの連続化が可能になる．このとき，不要なオブジェクトに添字を与えず振りなおすことにより，さらに配列内のデータ配置が局所化される．

4.3　メソッドに付加されたアノテーションに基づいてコンパイラが挿入するコード

メソッドに付加されたアノテーションに基づいて，コンパイラはメソッドの定義を変換し，メソッド呼び出し周辺にコードを挿入する．メソッドは前後処理を行うメソッドと元の処理を

ソースコード 4.9: メソッド定義変換の概要

```
1  // T: Target アノテーションが付与されたクラス
2  // listOfT: List<T>型の変数､再整列したい順序
3  // T.f1: Reserved アノテーションが付与されたフィールド
4
5  // 変換前メソッド定義
6  @Reorder(classes = {"T"}, orders={"listOfT"})
7  @AllocateFields({"T.f1"})
8  R function(P1 p1, P2 p2, ...){
9    /* 処理 */
10 }
11
12 // 変換後メソッド定義
13 R function$original(P1 p1, P2 p2, ...){
14   /* 処理 */
15 }
16 // 前後処理用メソッド
17 R function(P1 p1, P2 p2, ...){
18   // 再整列
19   int[] T_mapping_ = Reorderable.reorder(listOfT, T.data);
20   // field の再整列
21   T.fieldReorder(T_mappiing);
22   // AllocateFields で指定されたフィールドの確保
23   T.f1 = Reorderable.allocateArray(T.f1, T.data);
24   // 元の関数呼び出し
25   R _ret_ = function$original(p1, p2, ...);
26   // AllocateFields で指定されたフィールドの解放
27   T.f1 = null;
28   return ret;
29 }
```

行うメソッドに分割される．前後処理を行うメソッドを元のメソッド名とし，処理を行うメソッドの名前を変更する．メソッド定義変換の概要をソースコード 4.9 で示す．ダイクストラ法に適用したものをソースコード 4.1 の 28 行目から 35 行目で示している．

オブジェクトの再整列は Rotatable.reorder()(ソースコード 4.10) に，整列したい順序でオブジェクトを格納したリストを渡すことで行われる．reorder メソッドではではオブジェクトの再生成，Poitner オブジェクトの添字書き換えなどが行われている．フィールドの並び替えは，fieldReorder メソッドに添字変換配列を渡して行う．ソースコード 4.9 では，19 行目でオブジェクト再整列，21 行目でフィールド再整列のコードを挿入している．

メソッドローカルなフィールドは元のメソッド呼び出しの直前と直後に行う．Reorderable.allocateArray は第一引数に渡された変数の型と第二引数で与えた Util オブジェクトが管理する現在のオブジェクト数に応じて，配列を確保して返す．元のオブジェクトの呼び出しの直後に配列の解放を行うコードを挿入する．ソースコード 4.9 では，23 行目で配列化したフィールドのためのメモリ領域確保，27 行目でその配列の解放を行っている．

4.4　素朴な実装法

素朴な配列化の実装法はいくつか考えられるが，性能の向上し，ガベージコレクションや実行中のオブジェクト再整列可能である実装ではない．素朴な実装の二つの例をソースコード

ソースコード 4.10: Reorderable クラス定義

```
1  public abstract class Reorderable implements Cloneable{
2    public Pointer getNewInstance(Util data){
3      List<Reorderable> objList = data.objList;
4      List<List<WeakReference<Pointer>>> pointers = data.pointers;
5      int instanceId = pointers.size();
6      Pointer p = new Pointer(instanceId);
7      pointers.add(new ArrayList<WeakReference<Pointer>>());
8      pointers.get(instanceId).add(new WeakReference<Pointer>(p));
9      objList.add(this);
10     return p;
11   }
12   // 全てのプリミティブ型に同様に実装
13   static public int[] resizeArray(int[] b, Util data){
14     if(b == null || b.length != data.objList.size())return null;
15     int[] ret = new int[b.length * 2];
16     System.arraycopy(b, 0, ret, 0, b.length);
17     return ret;
18   }
19   // 全てのプリミティブ型に同様に実装
20   static public int[] allocateArray(int[] b, Util data){
21     return new int[data.objList.size()];
22   }
23   // 全てのプリミティブ型に同様に実装
24   public static int[] reorderFields(int[] order, int[] a){
25     if(a == null)return null;
26     int cnt = 0;
27     for(int i :order){
28       if(i != -1)cnt++;
29     }
30     int[] newA = new int[cnt];
31     for(int i = 0; i<order.length; i++){
32       if(order[i] == -1)continue;
33       newA[order[i]] = a[i];
34     }
35     return newA;
36   }
37   static public int[] reorder(List<Pointer> sortedList, Util data){
38     /* sortedList の格納順に基づいたオブジェクト再整列を行う。
39        sortedList 内の Pointer オブジェクトについても同様に再整列を行う。
40        再整列の結果に基づいて旧添字と現在の添字の対応表を返す。
41     */
42   }
43   public Reorderable clone(){
44       /* 略 */
45   }
46 }
```

2.6 とソースコード 4.11 で示す.

4.4.1 オブジェクトのフィールドとして固有の添字を追加する方法

各オブジェクトに固有の添字をフィールドとして追加する方法は，実行性能や配列化したフィールドのメモリ解放に問題が出てきてしまう．この実装法の利用例をソースコード 2.6 に，参照関係を図 4.4 に，フィールドアクセス時に必要な参照を図 4.5 に示す．ソースコード 2.6 の 5 行目のように元のオブジェクトの添字を格納するためのフィールドを追加し，2 行目のように静的配列として配列化対象のフィールドを管理する.

	手法 A	手法 B	手法 C
配列化したフィールドへのアクセスの向上	✓	✗	✓
オブジェクト再整列	✓	✗	✓
フィールド再整列	✓	✓	✗
オブジェクトのメモリの解放	✓	✓	✗
配列化したフィールドのメモリ解放	✓	✓	✗

表 4.1: 提案手法で採用した方法と素朴な実装法の比較

手法 A:　Pointer オブジェクトを使用した方法
手法 B:　オブジェクトのフィールドとして固有の添字を追加する方法
手法 C:　オブジェクトへの参照を添字を格納した整数型で置換する方法

配列化したフィールドへのアクセスは，配列化を行っていないフィールドへのアクセスよりも性能が悪くなる．図 4.5 で示した通り，この手法では配列化したフィールドへのアクセスには，添字を参照するためのオブジェクトアクセスが発生する．そのため，配列化を行わないフィールドへのアクセスの方が高速に行える．また，添字を格納しているもとオブジェクトのフィールドへのアクセスにより，配列化への連続的なメモリアクセスも阻害されてしまい，結果として配列化によるキャッシュヒット率の改善が見込めない．

オブジェクトの解放は変換前と変わらず行われるが，配列化したフィールドを格納するメモリ領域を解放するのは提案手法と類似した方法を取る必要がある．もしオブジェクトが不要となって Java のガベージコレクタによってメモリが解放されてもそのオブジェクトに対応するフィールドが配列からは解放されない．そのため，配列化したフィールドのメモリ管理は，ガベージコレクタとは別にユーザによって直接実装する必要が出てくる．Pointer オブジェクトを用いた方法では弱参照を用いて検出を行い，メモリ解放に利用している．他には検出には Finalizer を利用する方法が考えられる．

フィールドの再整列は行うことができるが，オブジェクト再生成によるオブジェクトの再整列を行うには，対象のオブジェクトの参照を持つ全ての変数を書き換える必要があり難しい．フィールドの再整列は各オブジェクトが持つ添字を書き換えることで実現可能である．行うためには各オブジェクトの参照を管理する必要がありガベージコレクタを阻害することが考えられるが，弱参照などを用いることにより解決可能である．しかしながら，オブジェクトの再整列のためにオブジェクトオブジェクトを再生成した場合，そのオブジェクトへの参照をもつ変数の書き換えが必要となる．この手法では参照関係が図 4.4 のように構築されているが，参照元の変数を書き換える必要が出てくる．Java の場合，ローカル変数の参照を持つことができない (ポインタがない) ため，変数を管理し書き換えることは難しい．

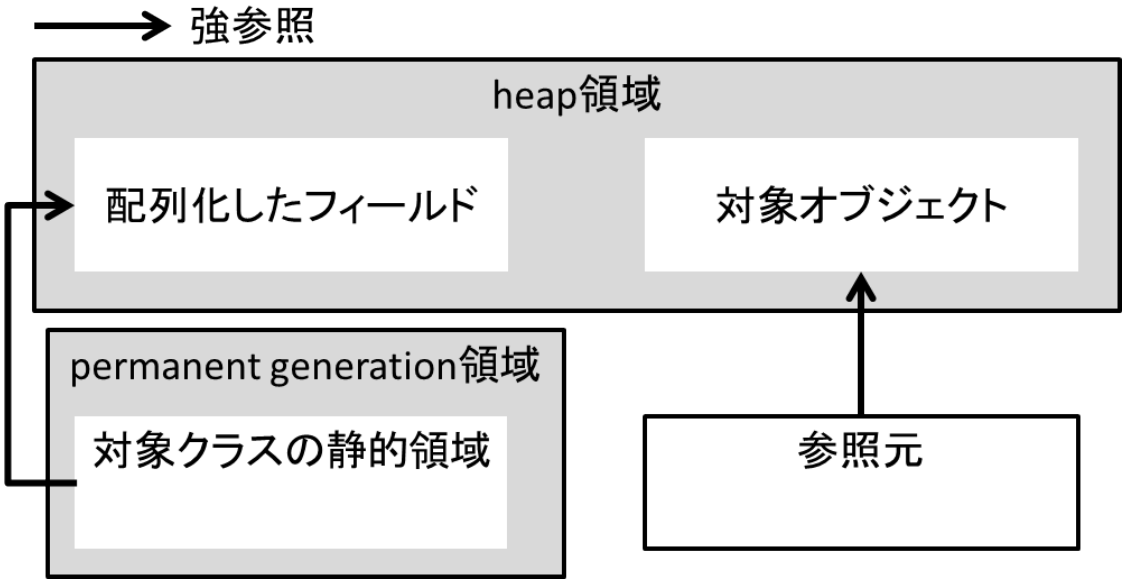

図 4.4: ソースコード 2.6 の実装における参照関係

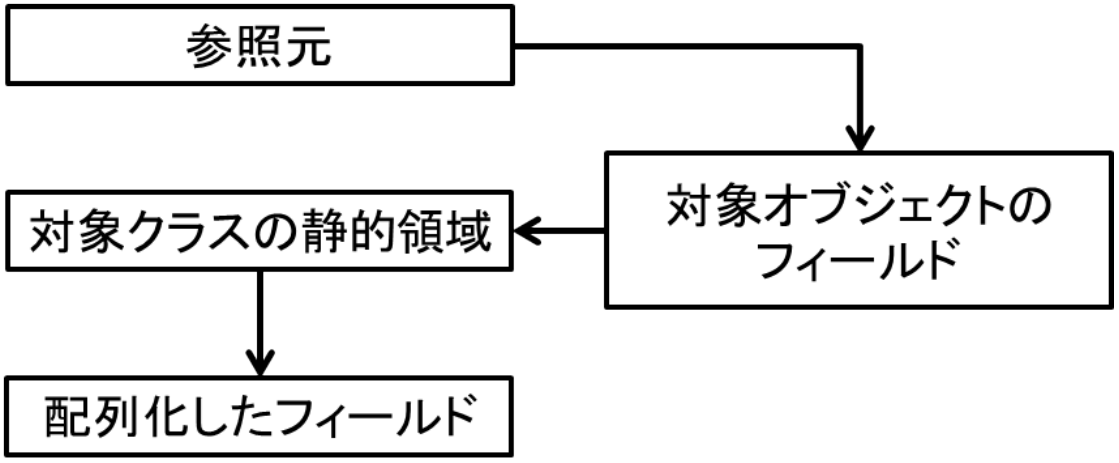

図 4.5: ソースコード 2.6 の実装におけるフィールドアクセス

4.4.2 オブジェクトへの参照を添字を格納した整数型で置換する方法

オブジェクトへの参照を添字を格納した整数型で置換する方法は間接参照のオーバーヘッドを低減させる事はできるが，付随的に新たなメモリ解放の問題が発生してしまう．この実装法の利用例をソースコード 4.11 に，参照関係を図 4.7 に，フィールドアクセス時に必要な参照を図 4.6 に示す．3 行目のように，Vertex オブジェクト全体を保持した配列 objList を，配列化した visited とは別に静的フィールドとして定義する．

配列化したフィールドへのアクセスはとても高速であり，元のオブジェクトへのアクセスも Pointer オブジェクトを利用した方法よりも高速に行える．図 4.6 で示した通り，配列化した

ソースコード 4.11: フィールド配列化の実装例 2

```
1  class Vertex{
2    public static boolean[] visitedList;
3    public static Vertex[] objList;
4    public int dist;
5    private Edge[] edges;
6    public Edge[] getEdges(){
7      return edges;
8    }
9  }
10
11 public void dijkstra(int s, Graph g){
12   // 非配列化フィールドへのアクセス
13   for(int v : g.getVertexList()) {
14     Vertex.objList[v].dist = inf;
15     Vertex.visitedList[v] = false;
16   }
17   Vertex.objList[s].dist = 0;
18   while(true){
19     int miniV = -1;
20     for(int v : g.getVertexList()){
21       // 配列化フィールドへのアクセス
22       if(Vertex.visitedList[v])continue;
23       if(miniV == -1 || Vertex.objList[miniV].dist > Vertex.objList[v].dist){
24         miniV = v;
25       }
26     }
27     if(miniV == -1)break;
28     Vertex.visitedList[miniV] = true;
29     // 略
30   }
31 }
```

フィールドを格納する配列を除けばオブジェクトアクセスを伴わずにデータにアクセスができる．しかし，オブジェクト全体を格納したリスト objList を全てのオブジェクトが共有するため，オブジェクト参照が切れる事は無く，新たにどのオブジェクトもガベージコレクタが解放できない問題も引き起こされる．さらに，添字を格納しているのは変数であり，ある特定のオブジェクトに対応した添字がプログラム中で持たれなくなること (参照されない不要なオブジェクトであること) を検知することは難しい．実現するには参照カウンタのようなガベージコレクタを自前で実装する必要がある．

オブジェクト再生成による再整列は行うことができるが，フィールドの再整列は対象のオブジェクトの添字を持つ全ての変数を書き換える必要があり難しい．オブジェクトの再生成による再整列は，オブジェクト全体を格納したリスト objList の参照先を書き換えることにより実現可能である．しかしながら，フィールドの再整列はそのフィールドを持つオブジェクトへの添字をもつ変数の書き換えが必要となる．この手法では参照関係が図 4.7 のように構築されているが，前節の手法と同様に，参照元の添字を格納する変数を書き換える必要がある．

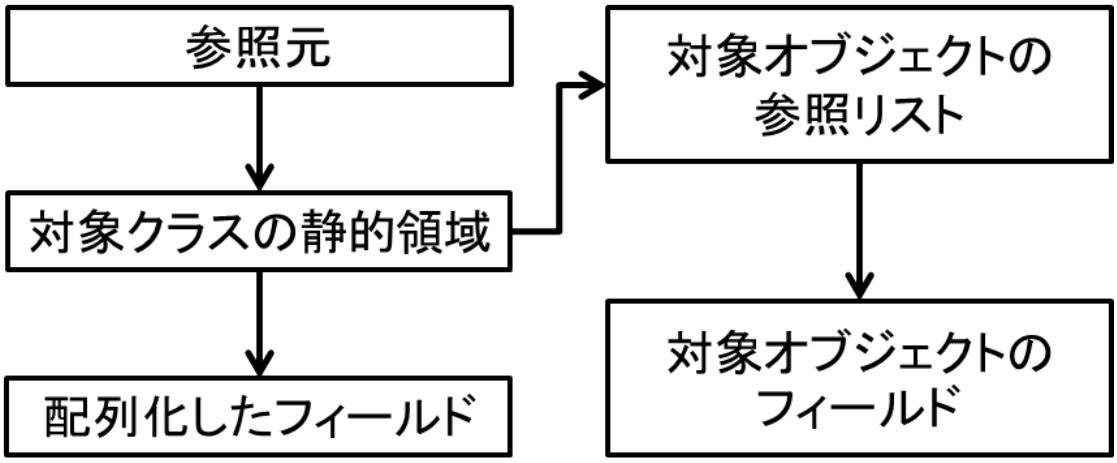

図 4.6: ソースコード 4.11 の実装におけるフィールドアクセス

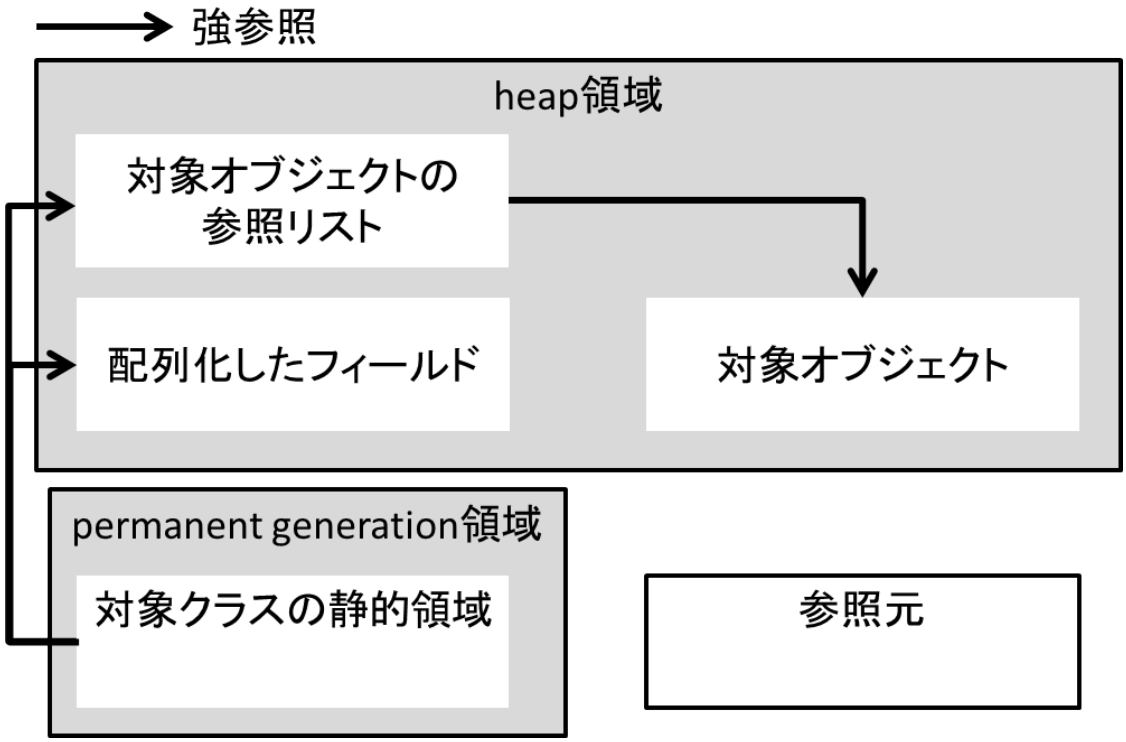

図 4.7: ソースコード 4.11 の実装における参照関係

第 5 章

実験

5.1 ダイクストラ法に対する最適化効果の実験の概要

提案した手法によるオブジェクトのメモリ内レイアウトの変更による実行性能の変化を確認するために，本研究で実装したコンパイラ (Javarac) で実行時間を測定した．ダイクストラ法を実装し，提案するフィールドの配列化とオブジェクトの再整列化適用した．ダイクストラ法は優先度付きキューなどを使用しない素朴な実装で，計算量は $O(V^2)$ である．一度の JavaVM の起動で一つのグラフを与え，64 回クエリを与えて各クエリの処理時間を測定した．各クエリはグラフの始点と終点のペアで，プログラムはその最短路の重みを出力する．処理時間は JIT コンパイラなどによる影響を考慮し，25 回目から 64 回目の 40 回分の平均と標準偏差を求めた．また，計測した処理時間が再整列化を時間を含み，ガベージコレクションが発生しないようダイクストラ法を行う直前に System.gc() を呼び出しガベージコレクタを促した．さらに，時間計測中にガベージコレクタによるプログラムの中断が起こっていないことも確かめた．

実験は以下の環境で行った．

- FUJITSU Supercomputer PRIMEHPC FX10 1 ノード
- Linux ベースの専用 OS (カーネル 2.6.25.8)
- SPARC64TM IXfx 1.848 GHz
- Memory 32 GB
- OpenJDK Runtime Environment (IcedTea6 1.11.5) (linux-gnu build 1.6.0 24-b24) OpenJDK 64-Bit Server VM (build 20.0-b12, mixed mode)
- 実行オプション -verbose:gc -Xms1600m -Xmx1600m (インクリメンタル GC 有効時 -Xincgc)

実験は二つの入力データに対して行った．入力データは重み付き無向グラフで疎なものと密なものを用意した．疎グラフは Graph500[29] ベンチマークで用いる入力データを使用した．Graph500 入力データは辺情報のみなので，[0, 99] の範囲の重みを乱数として与えて用いた．密グラフは TSPLIB[30] の二次元座標データ (dsj1000.tsp) を利用し，その座標データから完

全グラフを作成し実験を行った．入力データの情報は表 5.1 の通りである．

	密グラフ	疎グラフ
頂点数	1000	1024
辺数	$1000 * 999/2$	$1024 * 16$
元データ	dsj1000.tsp(TSPLIB)	Graph500 のジェネレータで生成

表 5.1: 入力データに用いたグラフ

プログラムはソースコード 2.5 のコードを基にアノテーションを付加して各最適化の有無や一部を改変し実験を行った．比較した要素は以下の通りである．

- どのフィールドを配列化するか．
- オブジェクト整列化を行うか，行う場合はどのような順か．
- 初期化時のオブジェクト生成順をダイクストラ法中のアクセス順序と一致させるか
- インクリメンタル GC を有効化するかどうか

配列化の有無はフィールド visited と dist に対して全通り試した．ソースコード 2.5 であるように，Vertex オブジェクトは visited(2 行目) と dist(3 行目) をフィールドとして持つ．この二つのフィールドの配列化の有無を以下のように全通りを試した．

- 配列化なし
- dist のみ配列化
- visited のみ配列化
- dist と visited の両方を配列化

再整列化は，参照順序に倣って行うことによる処理速度の向上を確かめるため，以下の通り試した．また，計測した処理時間が再整列化を時間を含む．

- 整列化なし
- 参照順にに沿って整列化
- 参照順序に沿わない整列化

5.2 考察

得られた結果について以下の点で考察を行った．

- オブジェクト再生成による効果
- フィールド配列化の効果
- 参照順序に沿った整列の効果
- コンパクションを伴う GC による整列順序の破壊

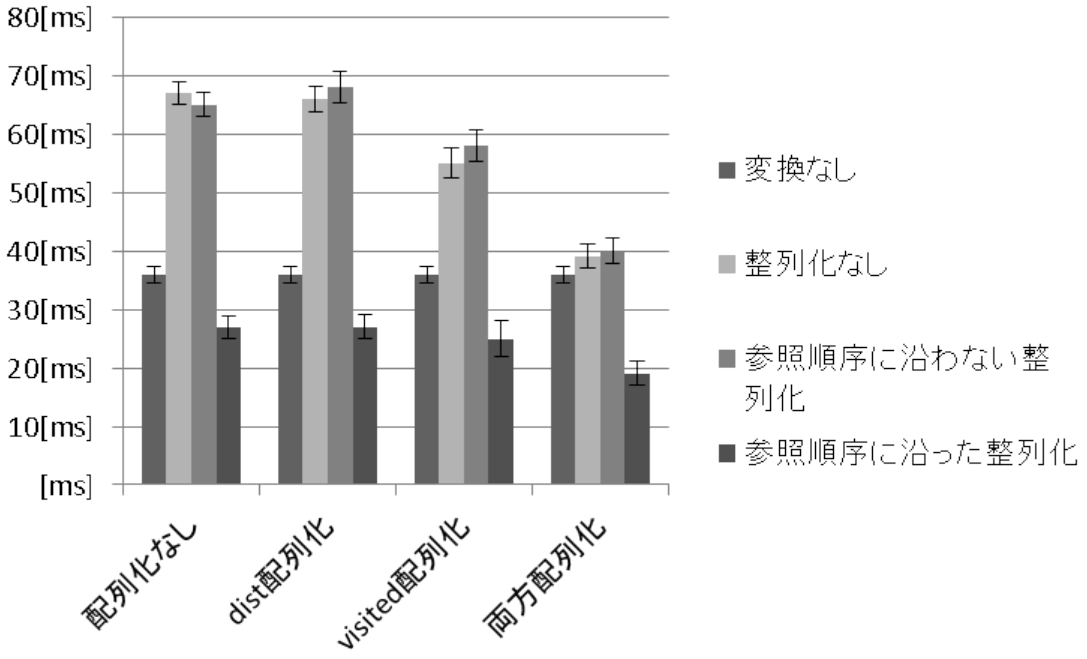

図 5.1: 疎グラフのダイクストラ法クエリ処理時間 (n=40)　初期オブジェクト順序と参照順序一致

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	36	—	—	—
整列化なし	67	66	55	39
参照順序に沿わない整列化	65	68	58	40
参照順序に沿った整列化	27	27	25	19

表 5.2: 疎グラフのダイクストラ法クエリ処理時間 平均値 (n=40) 初期オブジェクト順序と参照順序一致

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	1	—	—	—
整列化なし	2	2	3	2
参照順序に沿わない整列化	2	3	3	2
参照順序に沿った整列化	2	2	3	2

表 5.3: 疎グラフのダイクストラ法クエリ処理時間 標準偏差 (n=40) 初期オブジェクト順序と参照順序一致

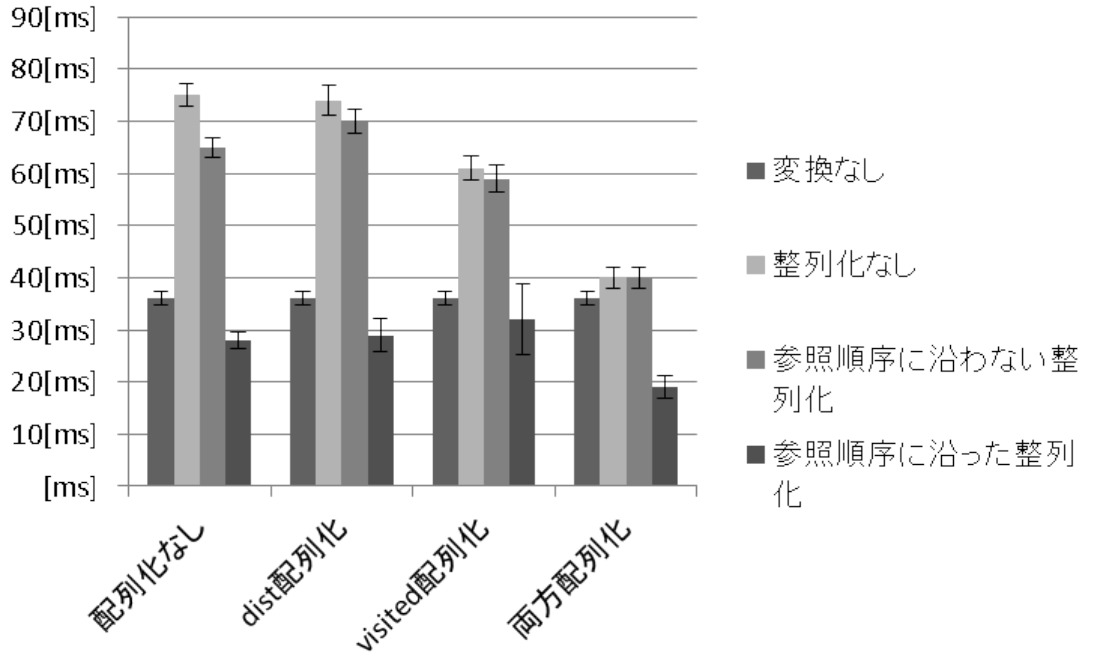

図 5.2: 疎グラフのダイクストラ法クエリ処理時間 (n=40) 初期オブジェクト順序ランダム

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	36	—	—	—
整列化なし	75	74	61	40
参照順序に沿わない整列化	65	70	59	40
参照順序に沿った整列化	28	29	32	19

表 5.4: 疎グラフのダイクストラ法クエリ処理時間 平均値 (n=40) 初期オブジェクト順序ランダム

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	1	—	—	—
整列化なし	2	3	2	2
参照順序に沿わない整列化	2	2	3	2
参照順序に沿った整列化	2	3	7	2

表 5.5: 疎グラフのダイクストラ法クエリ処理時間 標準偏差 (n=40) 初期オブジェクト順序ランダム

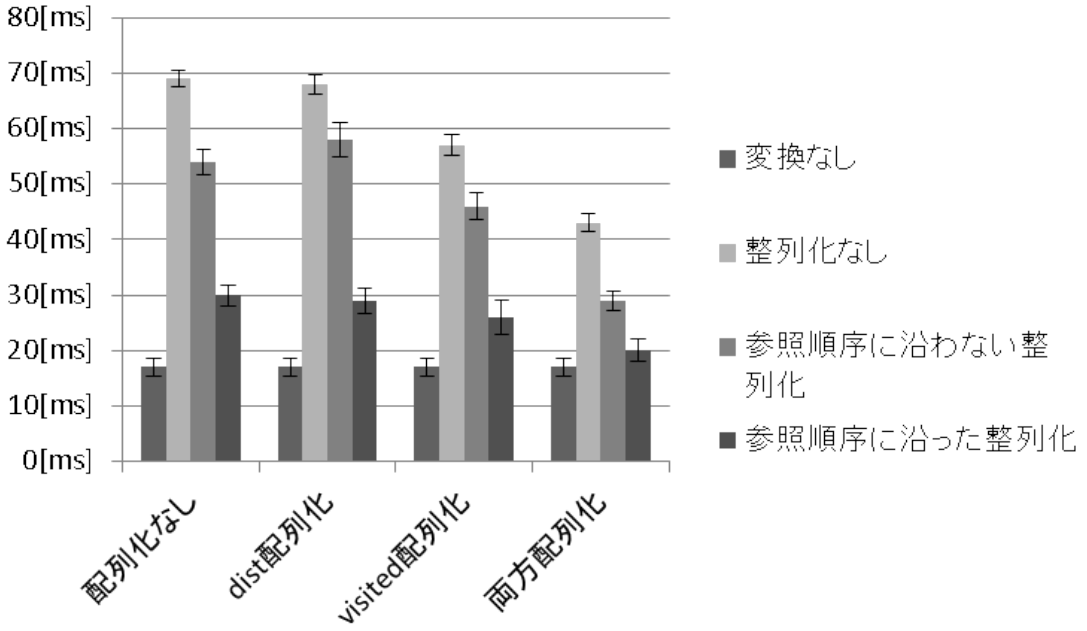

図 5.3: 疎グラフのダイクストラ法クエリ処理時間 (n=40)　初期オブジェクト順序と参照順序一致 インクリメンタル GC 有効

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	17	—	—	—
整列化なし	69	68	57	43
参照順序に沿わない整列化	54	58	46	29
参照順序に沿った整列化	30	29	26	20

表 5.6: 疎グラフのダイクストラ法クエリ処理時間 平均値 (n=40) 初期オブジェクト順序と参照順序一致 インクリメンタル GC 有効

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	2	—	—	—
整列化なし	2	2	2	2
参照順序に沿わない整列化	2	3	2	2
参照順序に沿った整列化	2	2	3	2

表 5.7: 疎グラフのダイクストラ法クエリ処理時間 標準偏差 (n=40) 初期オブジェクト順序と参照順序一致 インクリメンタル GC 有効

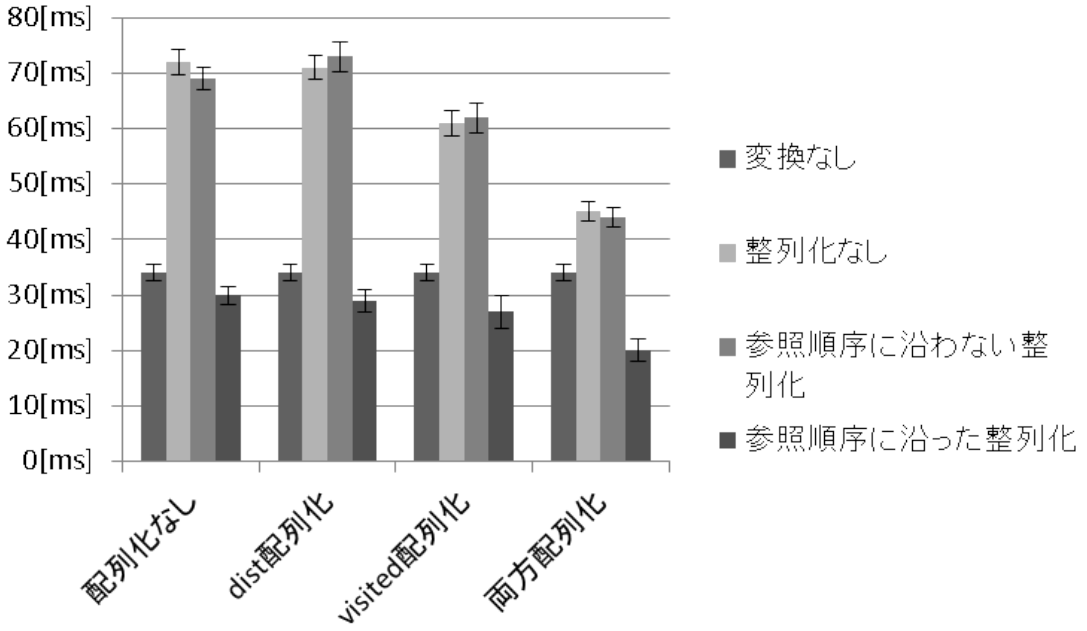

図 5.4: 疎グラフのダイクストラ法クエリ処理時間 (n=40)　初期オブジェクト順序ランダム インクリメンタル GC 有効

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	34	—	—	—
整列化なし	72	71	61	45
参照順序に沿わない整列化	69	73	62	44
参照順序に沿った整列化	30	29	27	20

表 5.8: 疎グラフのダイクストラ法クエリ処理時間 平均値 (n=40) 初期オブジェクト順序ランダム インクリメンタル GC 有効

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	1	—	—	—
整列化なし	2	2	2	2
参照順序に沿わない整列化	2	3	3	2
参照順序に沿った整列化	2	2	3	2

表 5.9: 疎グラフのダイクストラ法クエリ処理時間 標準偏差 (n=40) 初期オブジェクト順序ランダム インクリメンタル GC 有効

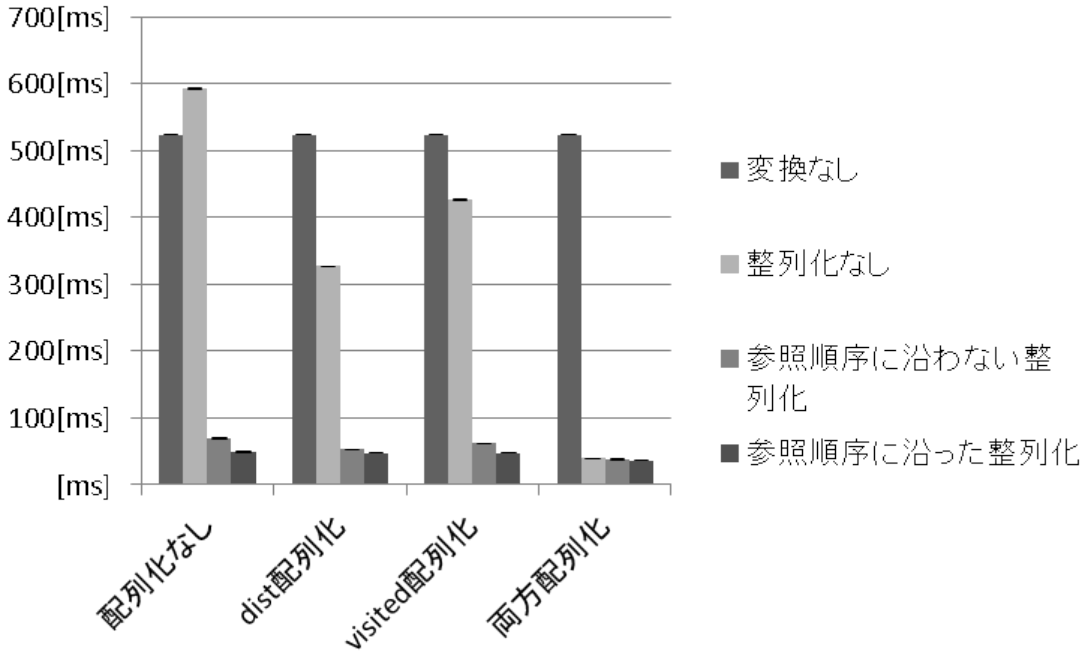

図 5.5: 密グラフのダイクストラ法クエリ処理時間 (n=40) 初期オブジェクト順序と参照順序一致

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	524	—	—	—
整列化なし	593	327	427	40
参照順序に沿わない整列化	69	53	62	38
参照順序に沿った整列化	49	48	48	37

表 5.10: 密グラフのダイクストラ法クエリ処理時間 平均値 (n=40) 初期オブジェクト順序と参照順序一致

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	0	—	—	—
整列化なし	0	0	1	0
参照順序に沿わない整列化	1	0	1	0
参照順序に沿った整列化	0	0	0	0

表 5.11: 密グラフのダイクストラ法クエリ処理時間 標準偏差 (n=40) 初期オブジェクト順序と参照順序一致

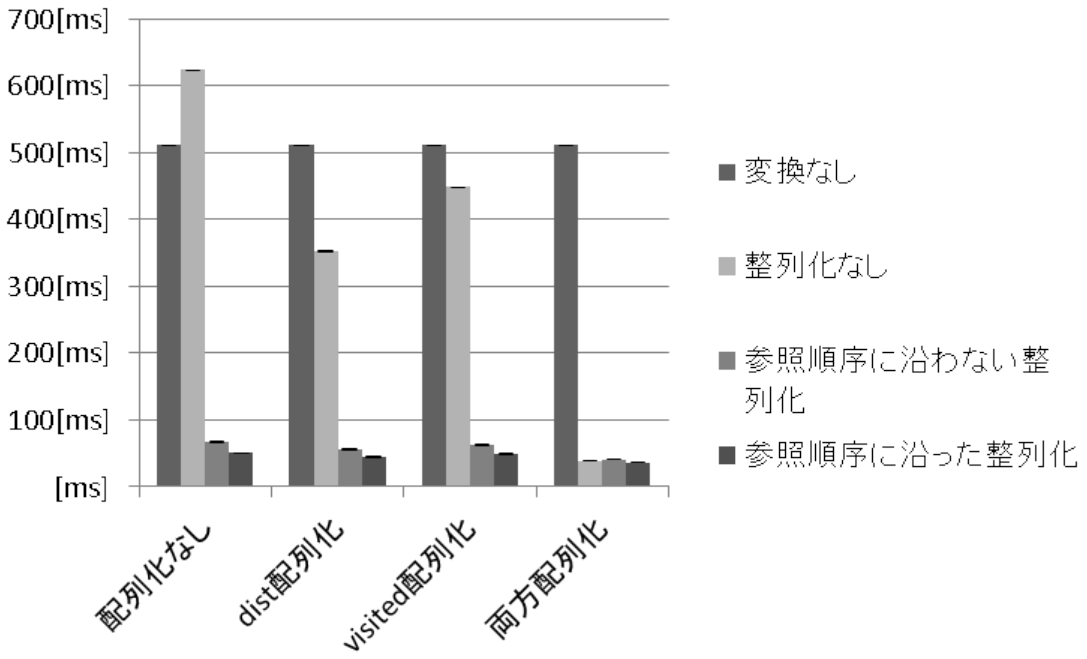

図 5.6: 密グラフのダイクストラ法クエリ処理時間 (n=40) 初期オブジェクト順序ランダム

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	511	—	—	—
整列化なし	623	352	448	39
参照順序に沿わない整列化	67	56	63	41
参照順序に沿った整列化	51	45	49	37

表 5.12: 密グラフのダイクストラ法クエリ処理時間 平均値 (n=40) 初期オブジェクト順序ランダム

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	0	—	—	—
整列化なし	0	0	1	0
参照順序に沿わない整列化	1	0	0	0
参照順序に沿った整列化	0	0	0	0

表 5.13: 密グラフのダイクストラ法クエリ処理時間 標準偏差 (n=40) 初期オブジェクト順序ランダム

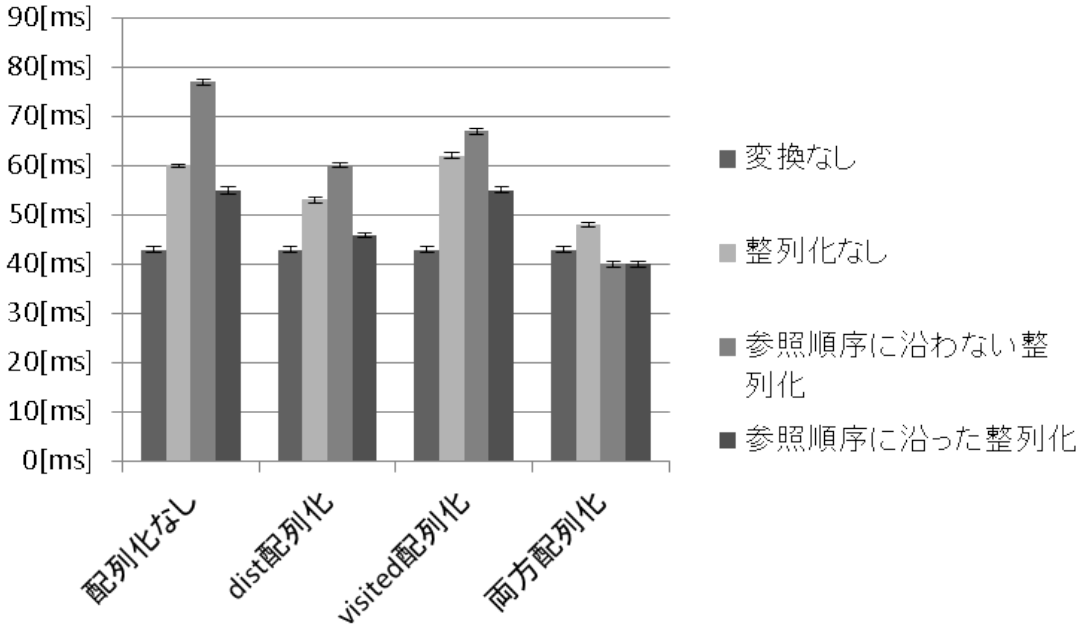

図 5.7: 密グラフのダイクストラ法クエリ処理時間 (n=40)　初期オブジェクト順序と参照順序一致 インクリメンタル GC 有効

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	43	—	—	—
整列化なし	60	53	62	48
参照順序に沿わない整列化	77	60	67	40
参照順序に沿った整列化	55	46	55	40

表 5.14: 密グラフのダイクストラ法クエリ処理時間 平均値 (n=40) 初期オブジェクト順序と参照順序一致 インクリメンタル GC 有効

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	0	—	—	—
整列化なし	0	0	1	0
参照順序に沿わない整列化	1	0	1	0
参照順序に沿った整列化	1	0	1	1

表 5.15: 密グラフのダイクストラ法クエリ処理時間 標準偏差 (n=40) 初期オブジェクト順序と参照順序一致

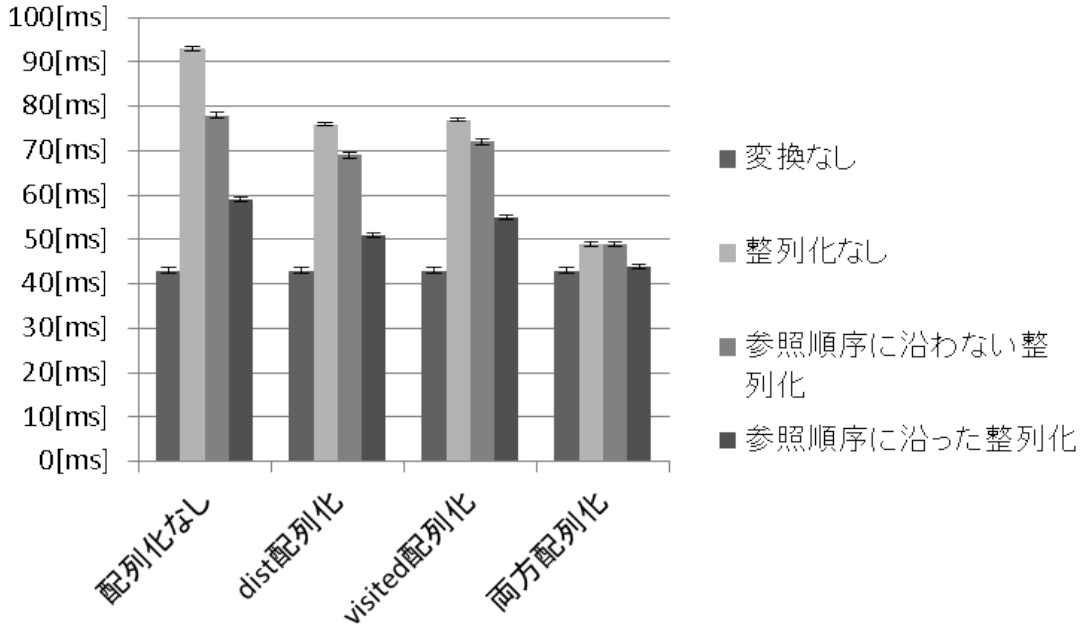

図 5.8: 密グラフのダイクストラ法クエリ処理時間 (n=40)　初期オブジェクト順序ランダム インクリメンタル GC 有効

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	43	—	—	—
整列化なし	93	76	77	49
参照順序に沿わない整列化	78	69	72	49
参照順序に沿った整列化	59	51	55	44

表 5.16: 密グラフのダイクストラ法クエリ処理時間 平均値 (n=40) 初期オブジェクト順序ランダム インクリメンタル GC 有効

	配列化なし	dist 配列化	visited 配列化	両方配列化
変換なし	1	—	—	—
整列化なし	0	0	0	0
参照順序に沿わない整列化	1	1	1	1
参照順序に沿った整列化	0	1	1	0

表 5.17: 密グラフのダイクストラ法クエリ処理時間 標準偏差 (n=40) 初期オブジェクト順序ランダム インクリメンタル GC 有効

メモリ上で多様なオブジェクトが混在している場合，生成順序にかかわらずアクセスしたいオブジェクトの再生成を行うことは大幅な速度改善になり，最大で約 12 倍の高速化 (表 5.12) が得られた．再生成のみの効果は参照順序に沿わない整列化を行った場合で，かつ，配列化を行ってない場合の結果に確認できる．密グラフでインクリメンタル GC を有効化していない結果 (図 5.5, 図 5.6) で 10 倍以上の高速化を確認できる．インクリメンタル GC を有効化すると最適化を行なっていないプログラムでも高速化することから，コンパクションを伴う GC などの発生により辺オブジェクトなどのほかのオブジェクトと混合されることにより参照の局所性が低下しているため，再生成による頂点オブジェクトをまとめて配置することの効果が大きいと考えられる．

フィールド配列化は大きく効果が出ており，最大で約 16 倍の高速化が観測された (表 5.12)．整列化の有無にかかわらずすべての場合で，dist, visited 両フィールドを配列化した場合実行時間が改善している．片方のフィールドのみの配列化では悪化している場合がある (表 5.4 など) があるは，これは同じオブジェクトであったため，まとめて配置されていた dist と visited が離れることから引き起こされていると考えられるが，両方を配列化した場合その恩恵が上回る．

参照順序に沿った整列も参照順序に沿わない整列化と比較して全ての場合で高速化されており，最大で約 2.5 倍程度 (表 5.8) の高速化がされた．この結果からメソッド呼び出し直前に順序に沿った整列化はメソッド内での参照速度に影響を与えること，処理が行われるまで順序が保持されたと考えられる．

インクリメンタル GC を有効化すると，初期化時のオブジェクト生成順序が結果 (表 5.8, 表 5.6 に大きく影響を与えた．これらは通常の GC が初期オブジェクト整列順序に依らない整列を dijkstra 法の呼び出しまでに行っていると考えられる．このことから，処理直前のオブジェクト整列化は重要である．

第 6 章

まとめと今後の課題

本研究ではアノテーションで指定したメソッド内で，オブジェクトのフィールドを配列化し再整列させられる Java コンパイラを提案した．ユーザはオブジェクト指向で記述されたソースコードに適切なアノテーションを付与するだけで，フィールドの配列化やオブジェクトの再整列化によって性能の向上が得られる．メモリ内レイアウトの切り替え法として，フィールド配列化とオブジェクト再整列化をサポートしている．

フィールド配列化とオブジェクト再整列化を実現する拡張 Java コンパイラ Javarac を開発し，実験によってその性能を確かめた．Javarac はユーザが付加したアノテーションにしたがって，静的にクラス定義やフィールドへのアクセス方法を変更し，メモリレイアウト変更のためのコードを挿入する．実験では，フィールド配列化とオブジェクト再整列化を段階的に切り替えることにより，フィールド配列化やオブジェクト再整列化の効果や，Javarac によるオブジェクト表現のオーバーヘッドを測定した．最も高速化した例では，通常の Java に比べて 10 倍以上高速化した．

一方，現状の Javarac の実装には以下のような制限がある．

- 分割コンパイルができない
- Target アノテーション付きクラスは，Cloneable であり，親クラスが Object でなければならず，子クラスを定義してはいけない
- Reorder 付きメソッド呼び出し時以外での整列化対象オブジェクトのメモリ解放ができない

これらの制限の解決は今後の課題である．

また，整列化対象クラスのオブジェクトは，オブジェクト再整列化を行うタイミング以外でオブジェクトを解放できない問題があった．ポインタオブジェクトが一定数解放されたらオブジェクトも解放したり，ユーザにアノテーションでタイミングを指定させるなどいくつかの解決法が考えられる．いくつかの方法をユーザに提示し，アノテーションのパラメータで選択できる実装を目標に検討していきたい．

さらに，より進んだメモリレイアウトの変換として，同じ型の複数のフィールドを配列化する際，一つの配列に格納する手法も考えられる．同様のレイアウトは C/C++ では構造体の

配列やポインタを利用して実現されている [24][23]．Java においては，より複雑な実現手段が必要と考えられるが，性能の向上が期待できるため合わせて取り組んでいきたい．

発表文献と研究活動

(1) 汐田 徹也, 山口 洋, 千葉 滋. アノテーションをもとにオブジェクトのデータレイアウトを自動的に変える Java コンパイラ. 第 15 回プログラミングおよびプログラミング言語ワークショップ PPL2013 2013 年 3 月 (ポスター)

(2) Tetsuya Shiota, Hiroshi Yamaguchi, Shigeru Chiba. Automatic Data Layout Reshaping in Object by Java Compiler. MODULARITY: aosd13 - Aspect-Oriented Software Development. March, 2013. (Poster)

参考文献

[1] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java (TM) Language Specification, The (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[2] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Addison-Wesley, 2013.

[3] Jgrapht. `http://jgrapht.org/`.

[4] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, Vol. 1, pp. 269–271, 1959.

[5] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146. ACM, 2010.

[6] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pp. 349–362, New York, NY, USA, 2012. ACM.

[7] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, Vol. 19, No. 2, pp. 248–264, April 1972.

[8] 秋葉拓哉, 岩田陽一, 北川宜稔. プログラミングコンテストチャレンジブック. 毎日コミュニケーションズ, 2010.

[9] jol. http://openjdk.java.net/projects/code-tools/jol/.

[10] Robert Strzodka. Data layout optimization for multi-valued containers in opencl. *Journal of Parallel and Distributed Computing*, Vol. 72, No. 9, pp. 1073 – 1082, 2012. Accelerators for High-Performance Computing.

[11] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pp. 13–24, New York, NY, USA, 1999. ACM.

[12] 前田昌樹, 鎌田十三郎, 瀧 和男. 共有メモリ型並列計算機におけるキャッシュを意識したオブジェクト内レイアウト法. 情報処理学会論文誌プログラミング (PRO), Vol. 42, No.

SIG03(PRO10), pp. 80–80, 2001.

[13] Michael Franz and Thomas Kistler. Splitting data objects to increase cache utilization. Technical report, 1998.

[14] D.N. Truong, Francois Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pp. 322–329, 1998.

[15] Ben Perry and Martin Swany. Improving mpi communication via data type fission. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pp. 352–355, New York, NY, USA, 2010. ACM.

[16] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference*. MIT press, 1995.

[17] Chris J Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, Vol. 13, No. 11, pp. 677–678, 1970.

[18] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pp. 235–246, New York, NY, USA, 1984. ACM.

[19] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, pp. 37–48, New York, NY, USA, 1998. ACM.

[20] Michael S Lam, Paul R Wilson, and Thomas G Moher. Object type directed garbage collection to improve locality. In *Memory Management*, pp. 404–425. Springer, 1992.

[21] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pp. 69–80, New York, NY, USA, 2004. ACM.

[22] Gene Novark, Trevor Strohman, and Emery D Berger. Custom object layout for garbage-collected languages. Technical report, Technical Report UM-CS-2006-06, UMass Amherst, 2006.

[23] Olga Golovanevsky and Ayal Zaks. Struct-reorg: current status and future perspectives. Proceedings of the GCC Developers' Summit, pp. 47–56, 2007.

[24] Peng Zhao, Shimin Cui, Yaoqing Gao, Raúl Silvera, and José Nelson Amaral. Forma: A framework for safe automatic array reshaping. *ACM Trans. Program. Lang. Syst.*, Vol. 30, No. 1, 2007.

[25] Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Di Xu. On-the-fly structure splitting for heap objects. *ACM Trans. Archit. Code Optim.*, Vol. 8, No. 4, pp. 26:1–26:20, January 2012.

[26] Aaftab Munshi, et al. The opencl specification. *Khronos OpenCL Working Group*, Vol. 1, pp. l1–15, 2009.

[27] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. *SIGPLAN Not.*, Vol. 42, No. 10, pp. 1–18, October 2007.

[28] Jesper Ö qvist and Gö rel Hedin. Extending the jastadd extensible java compiler to java 7. pp. 147–152. ACM, 2013.

[29] DA Bader, J Berry, S Kahan, R Murphy, J Riedy, and J Willcock. The graph 500 list: Graph 500 reference implementations. *Graph500. http://www.graph500.org/reference.html*, 2010.

[30] Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, Vol. 3, No. 4, pp. 376–384, 1991.

謝辞

本研究の方針や論文構成だけでなく、2 年間の研究活動を通して様々な面でご指導いただきました千葉滋教授に心より感謝致します。また、本研究の方針や実装、実験、論文執筆などについて詳細に指導していただいた東京大学情報基盤センターの佐藤芳樹特任講師に深く感謝致します。

また、研究を進める上での助言や技術的な支援を頂いた研究室の先輩の皆様に感謝致します。特に、1 年次の研究の進め方やプログラミングコンテスト活動を支えていただいた山口洋氏、多くの技術的な支援をいただいた武山文信氏に感謝致します。さらに修士課程在学中、様々な励ましや助言をいただいた同期の宗桜子さん、竹下若菜さん、穂積俊平さんに心から感謝致します。

最後に、多くの助言を頂いた千葉研究室の皆様に感謝致します。

付録 A

ソースコード

ソースコード A.1: Reorderable クラス定義

```
1  public abstract class Reorderable implements Cloneable{
2    public Pointer getNewInstance(Util data){
3      List<Reorderable> objList = data.objList;
4      List<List<WeakReference<Pointer>>> pointers = data.pointers;
5      int instanceId = pointers.size();
6      Pointer p = new Pointer(instanceId);
7      pointers.add(new ArrayList<WeakReference<Pointer>>());
8      pointers.get(instanceId).add(new WeakReference<Pointer>(p));
9      objList.add(this);
10     return p;
11   }
12   // 全てのプリミティブ型に同様に実装
13   static public int[] resizeArray(int[] b, Util data){
14     if(b == null || b.length != data.objList.size())return null;
15     int[] ret = new int[b.length * 2];
16     System.arraycopy(b, 0, ret, 0, b.length);
17     return ret;
18   }
19   // 全てのプリミティブ型に同様に実装
20   static public int[] allocateArray(int[] b, Util data){
21     return new int[data.objList.size()];
22   }
23   // 全てのプリミティブ型に同様に実装
24   public static int[] reorderFields(int[] order, int[] a){
25     if(a == null)return null;
26     int cnt = 0;
27     for(int i :order){
28       if(i != -1)cnt++;
29     }
30     int[] newA = new int[cnt];
31     for(int i = 0; i<order.length; i++){
32       if(order[i] == -1)continue;
33       newA[order[i]] = a[i];
34     }
35     return newA;
36   }
37   static public int[] reorder(List<Pointer> sortedList, Util data){
38     /* sortedList の格納順に基づいたオブジェクト再整列を行う。
39        sortedList 内の Pointer オブジェクトについても同様に再整列を行う。
40        再整列の結果に基づいて旧添字と現在の添字の対応表を返す。
41     */
42     List<Rotatable> oldObjList, newObjList;
43     List<List<WeakReference<Pointer>>> pointers, newPointers;
44     oldObjList = data.objList;
45     pointers = data.pointers;
```

```
46      final int oldSize = oldObjList.size();
47
48      int ret[] = new int[oldSize];
49      for(int i = 0; i<oldSize; i++)ret[i] = -1;
50      newObjList = new ArrayList<Rotatable>(oldSize);
51      newPointers = new ArrayList<List<WeakReference<Pointer>>>(oldSize);
52
53
54      for(Pointer i : sortedList){
55
56        int oldId = i.id, newId = newPointers.size();
57        ret[oldId] = newId;
58        newPointers.add(new ArrayList<WeakReference<Pointer>>());
59
60        for(WeakReference<Pointer> rp : pointers.get(oldId)){
61          Pointer p = rp.get();
62          if(p == null)continue;
63          p.id = newId;
64          newPointers.get(newId).add(rp);
65        }
66
67        pointers.set(oldId, null);
68
69        newObjList.add(oldObjList.get(oldId));
70      }
71
72      for(int i = 0; i<oldSize; i++){
73        if(pointers.get(i) == null)continue;
74
75        boolean allDeleted = true;
76        for(WeakReference<Pointer> rp : pointers.get(i)){
77          if(rp.get() != null)allDeleted = false;
78        }
79        if(allDeleted)continue;
80
81        int newId = newPointers.size();
82        ret[i] = newId;
83        newPointers.add(new ArrayList<WeakReference<Pointer>>());
84
85        for(WeakReference<Pointer> rp : pointers.get(i)){
86          Pointer p = rp.get();
87          if(p == null)continue;
88          p.id = newId;
89          newPointers.get(newId).add(rp);
90        }
91
92        pointers.set(i, null);
93        newObjList.add(oldObjList.get(i));
94      }
95
96      final int newSize = newObjList.size();
97
98      for(int i = 0; i<newSize; i++){
99        newObjList.set(i, newObjList.get(i).clone());
100     }
101
102     for(int i = 0; i<sortedList.size(); i++){
103       Pointer p = sortedList.get(i).clone();
104       sortedList.set(i, p);
105     }
106
107     for(int i = 0; i<sortedList.size(); i++){
108       newPointers.get(i).add(new WeakReference<Pointer>(sortedList.get(i)));
109     }
110
111     data.objList = newObjList;
112     data.pointers = newPointers;
```

```
113     return ret;
114   }
115   public Reorderable clone(){
116     try {
117       return (Rotatable) (super.clone());
118     } catch (CloneNotSupportedException e) {
119       throw new InternalError(e.toString());
120     }
121   }
122 }
```