情報通信工学専門実験 A
経路選択アルゴリズムの実装と評価

学籍番号：08D23091
氏名：辻 孝弥

2024 年 5 月 21 日実験

1　実験目的

本実験では、コンピュータネットワークにおける経路選択アルゴリズムを実装し、その性能を評価することを目的とする。具体的には、以下の 6 種類の経路選択アルゴリズムを実装し、呼損率という観点から性能を比較・評価する。

1. 最小ホップ経路を用いた固定経路（Dijkstra アルゴリズム）
2. 最大路を用いた固定経路
3. 最小ホップを用いた要求時経路
4. 最大路を用いた要求時経路
5. 空き容量の逆数を考慮した経路
6. 最短最大路（Shortest Widest Path）

また、指数分布に基づく通信モデルを導入し、より現実的なネットワーク環境でのシミュレーション評価を行う。

2　実験の理論的背景

2.1　経路選択アルゴリズム

2.1.1　最小ホップ経路（Dijkstra アルゴリズム）

Dijkstra アルゴリズムは、グラフ上の単一始点最短経路問題を解くためのアルゴリズムである。各ノード間の距離（ホップ数）を考慮し、始点から終点までの最短経路を求める。具体的には、未確定ノードの中から最短距離のノードを選び、そのノードを経由した場合の各ノードへの距離を更新していくという処理を繰り返す。

2.1.2 最大路アルゴリズム

最大路アルゴリズムは、経路上のボトルネックリンク（最も帯域幅が狭いリンク）の容量が最大となる経路を求めるアルゴリズムである。ネットワーク上の各リンクに容量があり、その容量が大きいリンクを優先的に使用することで、大きなデータ転送に適した経路を選択する。

2.1.3 要求時経路選択

要求時経路選択では、通信要求が発生した時点での空き容量に基づいて動的に経路を選択する。これにより、ネットワークの状態変化に適応した柔軟な経路選択が可能となる。

2.1.4 空き容量の逆数を考慮した経路選択

空き容量の逆数を考慮した経路選択では、リンクの重みを空き容量の逆数として扱う。空き容量が少ないリンクほど高いコストが設定され、混雑したリンクを避ける経路が選択される。

2.1.5 最短最大路（Shortest Widest Path）

最短最大路は、まず最大の帯域幅を持つ経路群を見つけ、その中から最もホップ数の少ない経路を選択するアルゴリズムである。これにより、十分な帯域を確保しつつ、効率的な経路を実現する。

2.2 評価指標

2.2.1 呼損率

呼損率は、全通信要求のうち、経路が確立できなかった通信の割合を表す。数式で表すと以下のようになる。

$$\text{呼損率} = \frac{\text{確立できなかった通信数}}{\text{全通信要求数}} \tag{1}$$

2.3 通信モデル

本実験では、指数分布に基づく通信モデルを採用した。通信の到着間隔と保持時間は指数分布に従って生成される。指数分布はランダムな事象の間隔を表すのに適しており、通信の発生パターンをより現実的にモデル化することができる。

3 実験結果

3.1 実装したアルゴリズムの動作

各経路選択アルゴリズムを実装し、10 ノードからなるネットワークでシミュレーションを行った。以下に各アルゴリズムの特徴と実装方法について述べる。

3.1.1 最小ホップ経路（Dijkstra）

始点から各ノードまでの最短距離を計算し、前ノード表を作成することで経路を特定する。未確定ノードの中から最短距離のノードを選ぶ処理を繰り返す実装とした。

3.1.2 最大路

リンクを容量の大きい順にソートし、大きな容量のリンクから順に追加していき、始点から終点への経路が見つかった時点でその経路を最大路として採用する。これにより、ボトルネックリンクの容量が最大の経路を見つけることができる。

3.1.3 要求時経路

通信要求があった時点でのネットワークの状態（リンクの空き容量）に基づいて経路を計算する。最小ホップ要求時経路では空き容量のあるリンクのみを使用して最短経路を計算し、最大路要求時経路では空き容量をリンクの重みとして最大路を計算する。

3.1.4 空き容量の逆数を考慮した経路

各リンクの重みを空き容量の逆数として設定し、Dijkstra アルゴリズムを適用する。これにより、空き容量の少ないリンクを避ける経路が選択される。

3.1.5 最短最大路

まず最大帯域幅を持つ経路を見つけ、その帯域幅以上の容量を持つリンクのみを使用して最短経路を計算する 2 段階のアプローチで実装した。

3.2 アルゴリズムの評価結果

図 1 に、各経路選択アルゴリズムにおけるパラメータ n と呼損率の関係を示す。パラメータ n は通信の持続時間に相当し、値が大きいほど長時間の通信を表す。

評価結果から、以下の点が観察された：

1. すべての経路選択方法において、パラメータ n が大きくなるほど呼損率が増加する傾向が見られた。
2. n=5 の短時間通信の場合、要求時経路選択（最小ホップ、最大路、空き容量の逆数、最短最大路）が非常に低い呼損率を示した。
3. n=50 以上の長時間通信では、固定経路選択（特に最小ホップ固定）が比較的良好な性能を示した。
4. 最小ホップ経路（固定）は全体的に安定した性能を示し、特に長時間通信においては最も低い呼損率を達成した。
5. 最大路（固定）は短時間通信では最小ホップ経路より性能が劣るが、帯域を重視する通信に

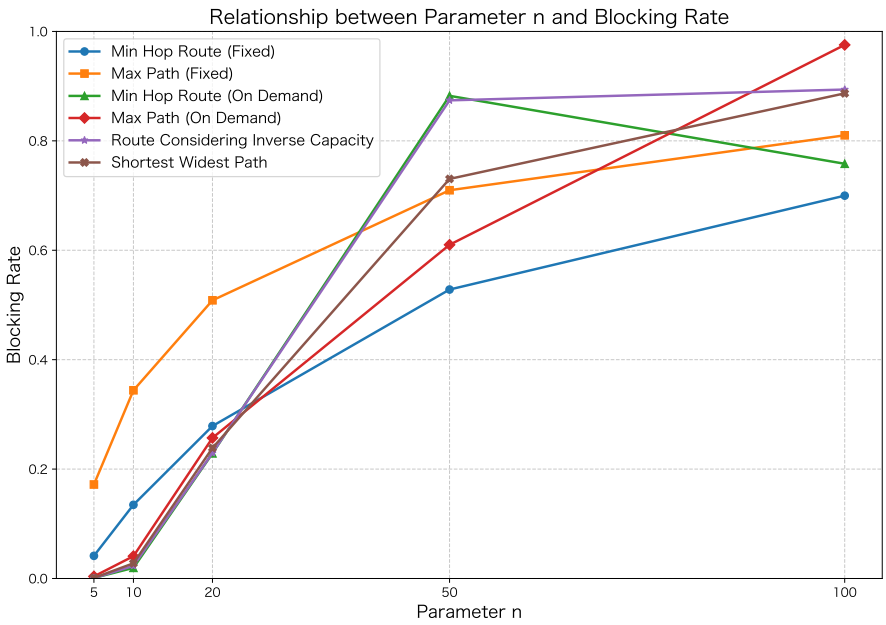

図 1　パラメータ n と呼損率の関係

は有効である。

4　考察

4.1　経路選択方法の性能比較

実験結果から、通信時間の長さによって最適な経路選択方法が異なることが分かった。

短時間通信（n=5, 10）の場合、要求時経路選択が非常に低い呼損率を実現した。これは、通信が短時間で終了するため、ネットワークの混雑が生じにくく、その時点での最適な経路を選択することが効果的だからと考えられる。特に、空き容量の逆数を考慮した経路と最短最大路は、n=5 において呼損率がほぼゼロに近い値を示しており、短時間通信に対して非常に効果的であることが確認された。

一方、長時間通信（n=50, 100）では、固定経路、特に最小ホップ経路（固定）が比較的良好な性能を示した。これは、長時間の通信が増えると、ネットワークの状態が頻繁に変化し、要求時に最適だった経路が時間の経過とともに最適でなくなる可能性があるためと考えられる。固定経路は、ネットワークトポロジに基づいた安定した経路を提供するため、長時間の通信に対してより堅牢な性能を示すと考えられる。

4.2　新規アルゴリズムの評価

本実験で新たに実装した「空き容量の逆数を考慮した経路」と「最短最大路」について考察する。

空き容量の逆数を考慮した経路は、短時間通信において非常に効果的であることが示された。空き容量が少ないリンクを避けることで、通信確立の成功率を高めることができた。特に n=5 では呼損率が 0.0006 と、ほぼすべての通信を確立することができた。

最短最大路も同様に短時間通信で良好な性能を示した。最大の帯域幅を確保しつつ最短経路を選択するという 2 段階のアプローチは、短時間通信において効果的に機能することが確認された。n=5 では呼損率が 0.0011 と極めて低い値を示した。

しかし、両アルゴリズムとも n=50, 100 のような長時間通信では呼損率が大きく上昇した。これは、通信時間が長くなると、高い帯域幅を要求する経路が長時間占有されるため、後続の通信要求に対応できなくなるためと考えられる。

4.3　指数分布による評価手法の意義

従来の等間隔評価に比べ、指数分布による評価手法を導入したことで、より現実的な通信パターンをシミュレーションすることができた。実際のネットワークでは、通信の到着時間や持続時間は一様分布ではなく、より確率的な性質を持つため、指数分布に基づくモデルはより現実に近い評価を可能にしたと考えられる。

この評価手法により、短時間通信と長時間通信が混在する実際のネットワーク環境でのアルゴリズムの性能をより正確に予測することができるようになった。

5　まとめ

本実験では、6 種類の経路選択アルゴリズムを実装し、指数分布に基づく通信モデルを用いてその性能を評価した。実験結果から、通信時間の長さによって最適な経路選択方法が異なることが明らかになった。

短時間通信では要求時経路選択（特に空き容量の逆数を考慮した経路と最短最大路）が効果的である一方、長時間通信では固定経路選択（特に最小ホップ経路）が比較的安定した性能を示すことが分かった。

実際のネットワークでは、短時間通信と長時間通信が混在するため、通信の特性に応じて適切な経路選択アルゴリズムを選択することが重要である。また、ネットワークの負荷状況や要求されるサービス品質（QoS）に応じて、経路選択の方針を適応的に変更することも効果的であると考えられる。

今後の課題としては、より大規模なネットワークでの評価や、実際のトラフィックパターンに基づいたシミュレーション、複数の経路選択アルゴリズムを組み合わせたハイブリッドアプローチの検討などが挙げられる。

6　参考文献

1. 情報通信工学専門実験 A 実験テキスト

付録 A　プログラムリスト

以下に、実装した Java プログラムのリストを示す。

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Random;
import java.util.Scanner;
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;
import java.util.Arrays;

public class dijkstra {
    /* 定数定義 */
    static final int NODE_NUM = 10;    /* 総ノード数 */
    static final int MAX = 9999;       /* 無限大に相当する数 */
    static final int FLAG = 1;         /* のテストの場合はに、シミュレーション評価を行う場合はにするDijkstra01 */
    static final int ROUTE_TYPE = 5;   /* 経路選択方法：最短路、最大路、要求時最短路、要求時最大路、空き容量逆数、最短最大路0=1=2=3=4=5= */
    static final int SIM_COUNT = 10000; /* シミュレーション回数 */
    static final int[] PARAM_N = {5, 10, 20, 50, 100}; /* テストするパラメータの値n */
    static final double LAMBDA = 1.0; /* 指数分布のレート（λ）パラメータ */
    static final String[] ROUTE_TYPE_NAMES = {
        "最小ホップ経路を用いた固定経路",
        "最大路を用いた固定経路",
        "最小ホップを用いた要求時経路",
        "最大路を用いた要求時経路",
        "空き容量の逆数を考慮した経路",
        "最短最大路（Shortest Widest ）Path"
    };

    /* 指数分布に基づく乱数生成メソッド */
    private static double generateExponentialRandom(Random random, double lambda) {
        // 指数分布の逆関数法を使用
        return -Math.log(1.0 - random.nextDouble()) / lambda;
    }

    /* 通信履歴を管理するクラス */
    static class Communication {
```

```
36          int src;            // 送信元ノード
37          int dest;           // 宛先ノード
38          List<Integer> path; // 経路上のノード
39          boolean established; // 通信が確立したかどうか
40          double arrivalTime; // 通信の到着時間
41          double holdingTime; // 通信の保持時間
42
43          public Communication(int src, int dest, double arrivalTime, double
        holdingTime) {
44              this.src = src;
45              this.dest = dest;
46              this.path = new ArrayList<>();
47              this.established = false;
48              this.arrivalTime = arrivalTime;
49              this.holdingTime = holdingTime;
50          }
51
52          // 経路情報を設定
53          public void setPath(int[] nodePath) {
54              this.path.clear();
55              int current = this.dest;
56              this.path.add(current);
57
58              while (current != this.src && current < NODE_NUM) {
59                  current = nodePath[current];
60                  if (current >= NODE_NUM) {
61                      // 経路が見つからなかった場合
62                      this.path.clear();
63                      break;
64                  }
65                  this.path.add(current);
66              }
67          }
68      }
69
70      /* リンク情報を格納するクラス */
71      static class Link implements Comparable<Link> {
72          int node1;
73          int node2;
74          int capacity;
75
76          public Link(int node1, int node2, int capacity) {
77              this.node1 = node1;
78              this.node2 = node2;
79              this.capacity = capacity;
80          }
81
82          @Override
83          public int compareTo(Link other) {
```

```
84              // 容量の大きい順にソート
85              return Integer.compare(other.capacity, this.capacity);
86          }
87      }
88
89      /* アルゴリズムによる最短路の計算Dijkstra */
90      public static void findShortestPath(int[][] graph, int[] path, int[] dist,
        int[] chk, int src, int dest) {
91          int i, tmp_node, tmp_dist, fin;
92
93          /* 初期化 */
94          for (i = 0; i < NODE_NUM; i++) {
95              dist[i] = MAX;
96              chk[i]  = 0;
97              path[i] = NODE_NUM;
98          }
99          path[src] = src;    /* 始点ノードへの経路上の前ノードはそれ自身とする */
100         dist[src] = 0;      /* 始点ノード自身への距離はである0 */
101         chk[src]  = 1;      /* 始点ノードへの最短距離は確定 */
102         tmp_node = src;     /* 始点ノードから探索を始める */
103         fin      = 0;
104
105         /* 経路探索 */
106         while (fin == 0) { /* フラグが立つまで繰り返すfin */
107             /* 更新処理 */
108             for (i = 0; i < NODE_NUM; i++) {
109                 if (graph[tmp_node][i] < MAX && chk[i] == 0) {
110                     /* 未確定ノードへの距離を更新 */
111                     if (dist[i] > dist[tmp_node] + graph[tmp_node][i]) {
112                         dist[i] = dist[tmp_node] + graph[tmp_node][i];
113                         path[i] = tmp_node;  /* 前ノードを記録 */
114                     }
115                 }
116             }
117
118             /* 次の最短距離ノードを確定 */
119             tmp_dist = MAX;
120             for (i = 0; i < NODE_NUM; i++) {
121                 if (chk[i] == 0 && dist[i] < tmp_dist) {
122                     tmp_dist = dist[i];
123                     tmp_node = i;
124                 }
125             }
126
127             // 未確定ノードが存在しない、または到達不能な場合 ループを抜ける
128             if (tmp_dist == MAX) {
129                 fin = 1;
130             } else {
131                 chk[tmp_node] = 1;  /* 最短距離ノードを確定 */
```

```
132             }
133
134             if (chk[dest] == 1) fin = 1; /* 終点ノードへの最短距離が確定したら終了 */
135         }
136     }
137
138     /* 深さ優先探索で経路を探す */
139     private static boolean dfs(int[][] graph, boolean[] visited, int[] path,
    int current, int dest) {
140         if (current == dest) {
141             return true;
142         }
143
144         visited[current] = true;
145
146         for (int next = 0; next < NODE_NUM; next++) {
147             if (graph[current][next] > 0 && !visited[next]) {
148                 path[next] = current;
149                 if (dfs(graph, visited, path, next, dest)) {
150                     return true;
151                 }
152             }
153         }
154
155         return false;
156     }
157
158     /* 最大路の計算（ボトルネックリンクが最大の経路を求める） */
159     public static void findMaximumPath(int[][] graph, int[][] link, int[] path,
     int[] bottleneck, int[] chk, int src, int dest) {
160         // リンクのリストを作成
161         List<Link> links = new ArrayList<>();
162         for (int i = 0; i < NODE_NUM; i++) {
163             for (int j = i + 1; j < NODE_NUM; j++) {
164                 if (link[i][j] > 0) {
165                     links.add(new Link(i, j, link[i][j]));
166                 }
167             }
168         }
169
170         // リンクを容量の大きい順にソート
171         Collections.sort(links);
172
173         // 経路探索用の一時グラフ
174         int[][] tempGraph = new int[NODE_NUM][NODE_NUM];
175         boolean[] visited = new boolean[NODE_NUM];
176         int currentBottleneck = 0;
177
178         // 容量の大きいリンクから順にグラフを構築
```

```
179         for (int i = 0; i < links.size(); i++) {
180             // 現在のリンクの容量をボトルネック容量の候補とする
181             currentBottleneck = links.get(i).capacity;
182 
183             // 同じ容量を持つリンクを全て追加
184             while (i < links.size() && links.get(i).capacity ==
    currentBottleneck) {
185                 Link currentLink = links.get(i);
186                 tempGraph[currentLink.node1][currentLink.node2] = 1;
187                 tempGraph[currentLink.node2][currentLink.node1] = 1;  // 無向グラ
    フなので両方向に追加
188                 i++;
189             }
190             i--; // ループで余分にインクリメントされた分を戻すwhile
191 
192             // 経路探索
193             Arrays.fill(visited, false);
194             Arrays.fill(path, NODE_NUM);
195             path[src] = src;
196 
197             if (dfs(tempGraph, visited, path, src, dest)) {
198                 // 経路が見つかった場合、この容量が最大ボトルネック容量
199                 for (int j = 0; j < NODE_NUM; j++) {
200                     bottleneck[j] = currentBottleneck;
201                 }
202                 return;
203             }
204         }
205 
206         // 経路が見つからなかった場合
207         Arrays.fill(bottleneck, 0);
208         Arrays.fill(path, NODE_NUM);
209     }
210 
211     /* 最小ホップを用いた要求時経路（空き容量のないリンクをグラフから取り除く） */
212     public static void findOnDemandShortestPath(int[][] graph, int[][]
    bandwidth, int[] path, int[] dist, int[] chk, int src, int dest) {
213         int i, j, tmp_node, tmp_dist, fin;
214         int[][] availableGraph = new int[NODE_NUM][NODE_NUM]; // 利用可能なリンクのみ
    のグラフ
215 
216         /* 利用可能なリンクのみのグラフを作成 */
217         for (i = 0; i < NODE_NUM; i++) {
218             for (j = 0; j < NODE_NUM; j++) {
219                 if (graph[i][j] < MAX && bandwidth[i][j] >= 1) {
220                     // 空き容量が1以上あるリンクのみを使用Mbps
221                     availableGraph[i][j] = graph[i][j];
222                 } else {
223                     availableGraph[i][j] = MAX;
```

```
                }
            }
        }

        /* 最短路の計算（利用可能なリンクのみ） */
        for (i = 0; i < NODE_NUM; i++) {
            dist[i] = MAX;
            chk[i] = 0;
            path[i] = NODE_NUM;
        }

        path[src] = src;
        dist[src] = 0;
        chk[src] = 1;
        tmp_node = src;
        fin = 0;

        while (fin == 0) {
            for (i = 0; i < NODE_NUM; i++) {
                if (availableGraph[tmp_node][i] < MAX && chk[i] == 0) {
                    if (dist[i] > dist[tmp_node] + availableGraph[tmp_node][i])
    {
                        dist[i] = dist[tmp_node] + availableGraph[tmp_node][i];
                        path[i] = tmp_node;
                    }
                }
            }

            tmp_dist = MAX;
            for (i = 0; i < NODE_NUM; i++) {
                if (chk[i] == 0 && dist[i] < tmp_dist) {
                    tmp_dist = dist[i];
                    tmp_node = i;
                }
            }

            if (tmp_dist == MAX) {
                fin = 1;
            } else {
                chk[tmp_node] = 1;
            }

            if (chk[dest] == 1) fin = 1;
        }
    }

    /* 最大路を用いた要求時経路（経路選択時点での空き容量をリンクの重みとする） */
    public static void findOnDemandMaximumPath(int[][] graph, int[][] bandwidth
    , int[] path, int[] bottleneck, int[] chk, int src, int dest) {
```

```
271         int i, j, tmp_node, tmp_bottleneck, fin;
272         int[][] availableGraph = new int[NODE_NUM][NODE_NUM]; // 利用可能なリンクのみのグラフ
273 
274         /* 利用可能なリンクのみのグラフを作成（空き容量を重みとする） */
275         for (i = 0; i < NODE_NUM; i++) {
276             for (j = 0; j < NODE_NUM; j++) {
277                 if (graph[i][j] < MAX && bandwidth[i][j] >= 1) {
278                     // 空き容量が1以上あるリンクのみを使用Mbps
279                     availableGraph[i][j] = bandwidth[i][j]; // 空き容量をリンクの重みとする
280                 } else {
281                     availableGraph[i][j] = 0;
282                 }
283             }
284         }
285 
286         /* 初期化 */
287         for (i = 0; i < NODE_NUM; i++) {
288             bottleneck[i] = 0;
289             chk[i] = 0;
290             path[i] = NODE_NUM;
291         }
292 
293         path[src] = src;
294         bottleneck[src] = MAX;
295         chk[src] = 1;
296         tmp_node = src;
297         fin = 0;
298 
299         while (fin == 0) {
300             for (i = 0; i < NODE_NUM; i++) {
301                 if (availableGraph[tmp_node][i] > 0 && chk[i] == 0) {
302                     int newBottleneck = Math.min(bottleneck[tmp_node], availableGraph[tmp_node][i]);
303 
304                     if (bottleneck[i] < newBottleneck) {
305                         bottleneck[i] = newBottleneck;
306                         path[i] = tmp_node;
307                     }
308                 }
309             }
310 
311             tmp_bottleneck = 0;
312             tmp_node = -1;
313             for (i = 0; i < NODE_NUM; i++) {
314                 if (chk[i] == 0 && bottleneck[i] > tmp_bottleneck) {
315                     tmp_bottleneck = bottleneck[i];
316                     tmp_node = i;
```

```
317                 }
318             }
319 
320             if (tmp_node == -1) {
321                 fin = 1;
322             } else {
323                 chk[tmp_node] = 1;
324             }
325 
326             if (chk[dest] == 1) fin = 1;
327         }
328     }
329 
330     /* 空き容量の逆数を考慮した経路選択 */
331     public static void findInverseCapacityPath(int[][] graph, int[][] bandwidth
    , int[] path, int[] dist, int[] chk, int src, int dest) {
332         int i, tmp_node, tmp_dist, fin;
333         double[][] inverseGraph = new double[NODE_NUM][NODE_NUM]; // 空き容量の逆数
    を格納するグラフ
334 
335         /* 空き容量の逆数でグラフを初期化 */
336         for (i = 0; i < NODE_NUM; i++) {
337             for (int j = 0; j < NODE_NUM; j++) {
338                 if (graph[i][j] < MAX && bandwidth[i][j] >= 1) {
339                     // 空き容量が1以上あるリンクのみを使用Mbps
340                     inverseGraph[i][j] = 1.0 / bandwidth[i][j]; // 空き容量の逆数を
    重みとする
341                 } else {
342                     inverseGraph[i][j] = MAX; // 使用できないリンク
343                 }
344             }
345         }
346 
347         /* 初期化 */
348         for (i = 0; i < NODE_NUM; i++) {
349             dist[i] = MAX;
350             chk[i] = 0;
351             path[i] = NODE_NUM;
352         }
353 
354         path[src] = src;
355         dist[src] = 0;
356         chk[src] = 1;
357         tmp_node = src;
358         fin = 0;
359 
360         /* アルゴリズムで最小コスト経路を探索Dijkstra */
361         while (fin == 0) {
362             for (i = 0; i < NODE_NUM; i++) {
```

```
                if (inverseGraph[tmp_node][i] < MAX && chk[i] == 0) {
                    int newDist = (int)(dist[tmp_node] + inverseGraph[tmp_node
][i] * 1000); // 小数を整数に
変換
                    if (dist[i] > newDist) {
                        dist[i] = newDist;
                        path[i] = tmp_node;
                    }
                }
            }

            tmp_dist = MAX;
            for (i = 0; i < NODE_NUM; i++) {
                if (chk[i] == 0 && dist[i] < tmp_dist) {
                    tmp_dist = dist[i];
                    tmp_node = i;
                }
            }

            if (tmp_dist == MAX) {
                fin = 1;
            } else {
                chk[tmp_node] = 1;
            }

            if (chk[dest] == 1) fin = 1;
        }
    }

    /* 最短最大路（Shortest Widest ）の計算Path */
    public static void findShortestWidestPath(int[][] graph, int[][] bandwidth,
     int[] path, int[] dist, int[] chk, int src, int dest) {
        int i, j;
        int[] bottleneck = new int[NODE_NUM];
        int maxBottleneck;

        // Step 1: 最大帯域幅を持つ経路を見つける
        findOnDemandMaximumPath(graph, bandwidth, path, bottleneck, chk, src,
dest);
        maxBottleneck = bottleneck[dest];

        // 経路が見つからない場合は終了
        if (maxBottleneck == 0) {
            Arrays.fill(path, NODE_NUM);
            Arrays.fill(dist, MAX);
            return;
        }

        // Step 2: 最大帯域幅以上の容量を持つリンクのみを使用して最短経路を計算
```

```
408         int[][] filteredGraph = new int[NODE_NUM][NODE_NUM];
409         for (i = 0; i < NODE_NUM; i++) {
410             for (j = 0; j < NODE_NUM; j++) {
411                 if (graph[i][j] < MAX && bandwidth[i][j] >= maxBottleneck) {
412                     // 最大帯域幅以上の容量を持つリンクのみを使用
413                     filteredGraph[i][j] = graph[i][j];
414                 } else {
415                     filteredGraph[i][j] = MAX;
416                 }
417             }
418         }
419 
420         // 最短経路を計算
421         findShortestPath(filteredGraph, path, dist, chk, src, dest);
422     }
423 
424     public static void main(String[] args) {
425         /* のアルゴリズム部分で必要な変数Dijkstra */
426         int[][] graph = new int[NODE_NUM][NODE_NUM];     /* 距離行列 */
427         int[] path    = new int[NODE_NUM];               /* 前ノード表 */
428         int[] dist    = new int[NODE_NUM];               /* 距離を格納 */
429         int[] bottleneck = new int[NODE_NUM];            /* ボトルネック容量を格納 */
430         int[] chk     = new int[NODE_NUM];               /* 最短距離確定のフラグ */
431         int tmp_node, tmp_dist;                          /* 注目しているノードとそこまで
    の最短距離 */
432         int src = 0, dest = 0;                           /* 始点・終点ノード */
433         int a = 0, b = 0, c = 0, d = 0, i = 0, j = 0;
434         int fin;                                         /* 未確定ノードが残っているかど
    うかのフラグ */
435 
436         /* シミュレーション評価の部分で必要な変数 */
437         int[][] link      = new int[NODE_NUM][NODE_NUM]; /* リンク容量 */
438         int[][] bandwidth = new int[NODE_NUM][NODE_NUM]; /* リンクの空き容量 */
439         int miss;                                        /* 呼損を表すフラグ */
440         int success;                                     /* 確立できた通信回数 */
441         int sum_success;                                 /* 確立できた通信回数の合計 */
442         int sim_time;                                    /* 評価の回数をカウント */
443         List<Communication> history = new ArrayList<>(); /* 通信履歴 */
444 
445         /*
446          * 距離行列の作成
447          */
448         for (i = 0; i < NODE_NUM; i++) {
449             for (j = 0; j < NODE_NUM; j++) {
450                 graph[i][j] = MAX;     /* 接続されていないノード間の距離をにするMAX */
451                 link[i][j]  = -1;      /* 接続されていないノード間のリンク容量をにする-1 */
452                 if (i == j) {
453                     graph[i][j] = 0;
454                     link[i][j]  = -1;
```

```
455                     } /* そのノード自身への距離はとし、リンク容量はとする0-1 */
456                 }
457             }
458
459             /* ファイル読み込み */
460             try {
461                 Scanner scanner = new Scanner(new File("./distance.txt"));
462                 while (scanner.hasNextInt()) { /* までつ組を読み込むEOF4 */
463                     a = scanner.nextInt();
464                     b = scanner.nextInt();
465                     c = scanner.nextInt();
466                     d = scanner.nextInt();
467                     graph[a][b] = c; /* 接続されているノード間の距離を設定 */
468                     graph[b][a] = c; /* 逆方向も等距離と仮定 */
469                     link[a][b]  = d; /* 接続されているノード間のリンクを設定 */
470                     link[b][a]  = d; /* 逆方向も同じ容量と仮定 */
471                 }
472                 scanner.close();
473             } catch (FileNotFoundException e) {
474                 e.printStackTrace();
475                 return;
476             }
477
478             /*
479              * 始点・終点ノードを標準入力から得る 評価の場合は、実行しない()
480              */
481             if (FLAG == 0) {
482                 Scanner stdin = new Scanner(System.in);
483                 System.out.printf("Source Node?(0-%d)", NODE_NUM - 1);
484                 src = stdin.nextInt();
485                 System.out.printf("Destination Node?(0-%d)", NODE_NUM - 1);
486                 dest = stdin.nextInt();
487                 // stdin.close();
488             }
489
490             if (FLAG == 1) {
491                 Random rand = new Random(System.currentTimeMillis()); /* 乱数の初期
    化 */
492
493                 // 各経路選択方法でシミュレーションを実行
494                 for (int route_type = 0; route_type < 6; route_type++) {
495                     System.out.printf("\n=======================================\n"
    );
496                     System.out.printf("経路選択方
    法: %s\n", ROUTE_TYPE_NAMES[route_type]);
497                     System.out.printf("=======================================\n");
498
499                     // パラメータごとに実験を実施n
500                     for (int param_n : PARAM_N) {
```

```
501                     System.out.printf("\n===== パラメータn = %d の実験結果 =====\n", param_n);
502
503                     success = 0;
504                     sum_success = 0; /* 評価指標を初期化 */
505                     history.clear(); /* 通信履歴をクリア */
506
507                     // 初期化
508                     for (i = 0; i < NODE_NUM; i++) {
509                         for (j = 0; j < NODE_NUM; j++) {
510                             bandwidth[i][j] = link[i][j];  // リンク容量を初期状態にコピー
511                         }
512                     }
513
514                     final double[] currentTime = {0.0}; // 現在時刻（指数分布用）
515
516                     // 回のシミュレーションSIM_COUNT
517                     for (sim_time = 0; sim_time < SIM_COUNT; sim_time++) {
518                         // 到着間隔を指数分布に基づいて生成
519                         double interArrivalTime = generateExponentialRandom(rand, LAMBDA);
520                         currentTime[0] += interArrivalTime;
521
522                         // ランダムに送受信ノードを決定
523                         do {
524                             src = rand.nextInt(NODE_NUM);
525                             dest = rand.nextInt(NODE_NUM);
526                         } while (src == dest);
527
528                         // 通信保持時間を指数分布に基づいて生成
529                         double holdingTime = generateExponentialRandom(rand, 1.0 / param_n); // パラメータに応じた平均保持時間n
530
531                         // 経路選択方法に応じた経路計算
532                         switch (route_type) {
533                             case 0: // 最小ホップ経路を用いた固定経路
534                                 findShortestPath(graph, path, dist, chk, src, dest);
535                                 break;
536                             case 1: // 最大路を用いた固定経路
537                                 findMaximumPath(graph, link, path, bottleneck, chk, src, dest);
538                                 break;
539                             case 2: // 最小ホップを用いた要求時経路
540                                 findOnDemandShortestPath(graph, bandwidth, path, dist, chk, src, dest);
541                                 break;
```

```
542                             case 3: // 最大路を用いた要求時経路
543                                 findOnDemandMaximumPath(graph, bandwidth, path,
    bottleneck, chk, src, dest);
544                                 break;
545                             case 4: // 空き容量の逆数を考慮した経路
546                                 findInverseCapacityPath(graph, bandwidth, path,
    dist, chk, src, dest);
547                                 break;
548                             case 5: // 最短最大路 (Shortest Widest ) Path
549                                 findShortestWidestPath(graph, bandwidth, path,
    dist, chk, src, dest);
550                                 break;
551                         }
552
553                         // 通信オブジェクトを作成
554                         Communication comm = new Communication(src, dest,
    currentTime[0], holdingTime);
555                         comm.setPath(path); // 経路情報を設定
556
557                         // 経路が見つからなかった場合はスキップ
558                         if (comm.path.isEmpty()) {
559                             history.add(comm);
560                             continue;
561                         }
562
563                         // 経路上のリンク容量をチェック
564                         boolean canEstablish = true;
565                         for (int idx = 0; idx < comm.path.size() - 1; idx++) {
566                             int node1 = comm.path.get(idx);
567                             int node2 = comm.path.get(idx + 1);
568                             if (bandwidth[node1][node2] < 1) {
569                                 canEstablish = false;
570                                 break;
571                             }
572                         }
573
574                         if (canEstablish) {
575                             // 通信を確立できる場合、経路上のリンク容量を1減少Mbps
576                             for (int idx = 0; idx < comm.path.size() - 1; idx
    ++) {
577                                 int node1 = comm.path.get(idx);
578                                 int node2 = comm.path.get(idx + 1);
579                                 bandwidth[node1][node2]--;
580                                 bandwidth[node2][node1]--;
581                             }
582                             comm.established = true;
583                             success++;
584                         }
585
```

```
586                         // 通信履歴に追加
587                         history.add(comm);
588 
589                         // 終了した通信のリンク容量を回復
590                         List<Communication> endedCommunications = new ArrayList
    <>();
591                         for (Communication oldComm : history) {
592                             if (oldComm.established &&
593                                 oldComm.arrivalTime + oldComm.holdingTime <=
    currentTime[0]) {
594                                 // 通信終了時刻が現在時刻以前なら終了
595                                 for (int idx = 0; idx < oldComm.path.size() -
    1; idx++) {
596                                     int node1 = oldComm.path.get(idx);
597                                     int node2 = oldComm.path.get(idx + 1);
598                                     bandwidth[node1][node2]++;
599                                     bandwidth[node2][node1]++;
600                                 }
601                                 endedCommunications.add(oldComm);
602                             }
603                         }
604                         // 終了した通信を履歴から削除
605                         history.removeAll(endedCommunications);
606                     }
607 
608                     // 呼損率を計算
609                     double blockingRate = (double)(SIM_COUNT - success) /
    SIM_COUNT;
610                     System.out.printf("確立できた通信
    数: %d / %d\n", success, SIM_COUNT);
611                     System.out.printf("呼損率: %.4f\n", blockingRate);
612                 }
613             }
614 
615             return; // シミュレーション終了
616         }
617 
618         /*
619          * シミュレーション評価の結果出力
620          */
621         System.out.printf("\naverage = %f\n", sum_success / 1000.0); /* 平均を表
    示 */
622     }
623 }
```

Listing 1　dijkstra.java