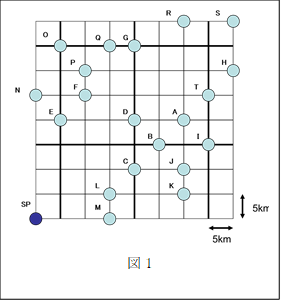
2018年5月30日

配送計画問題レポート

高度情報演習1A

岩本 裕大

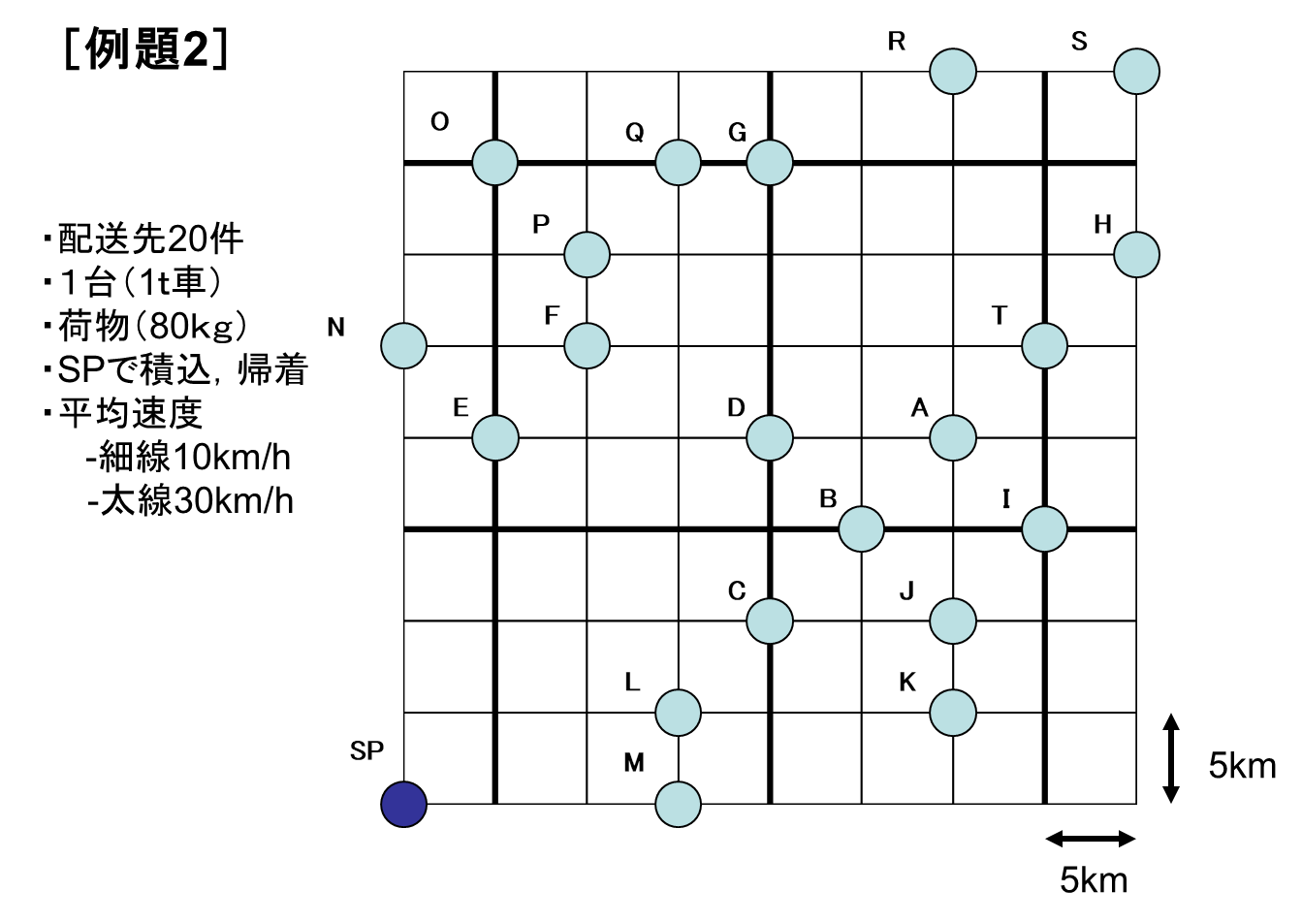
AL16014

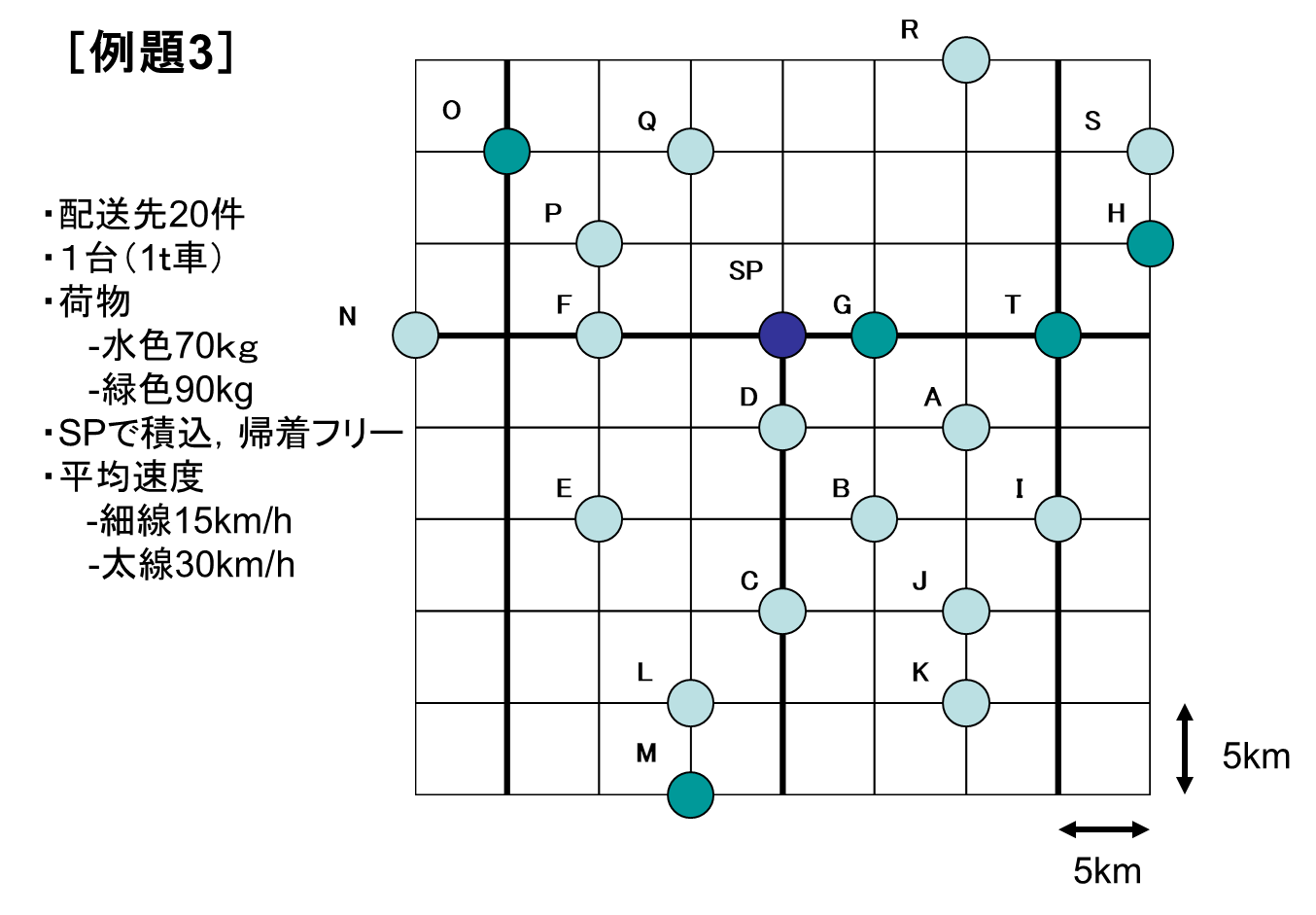
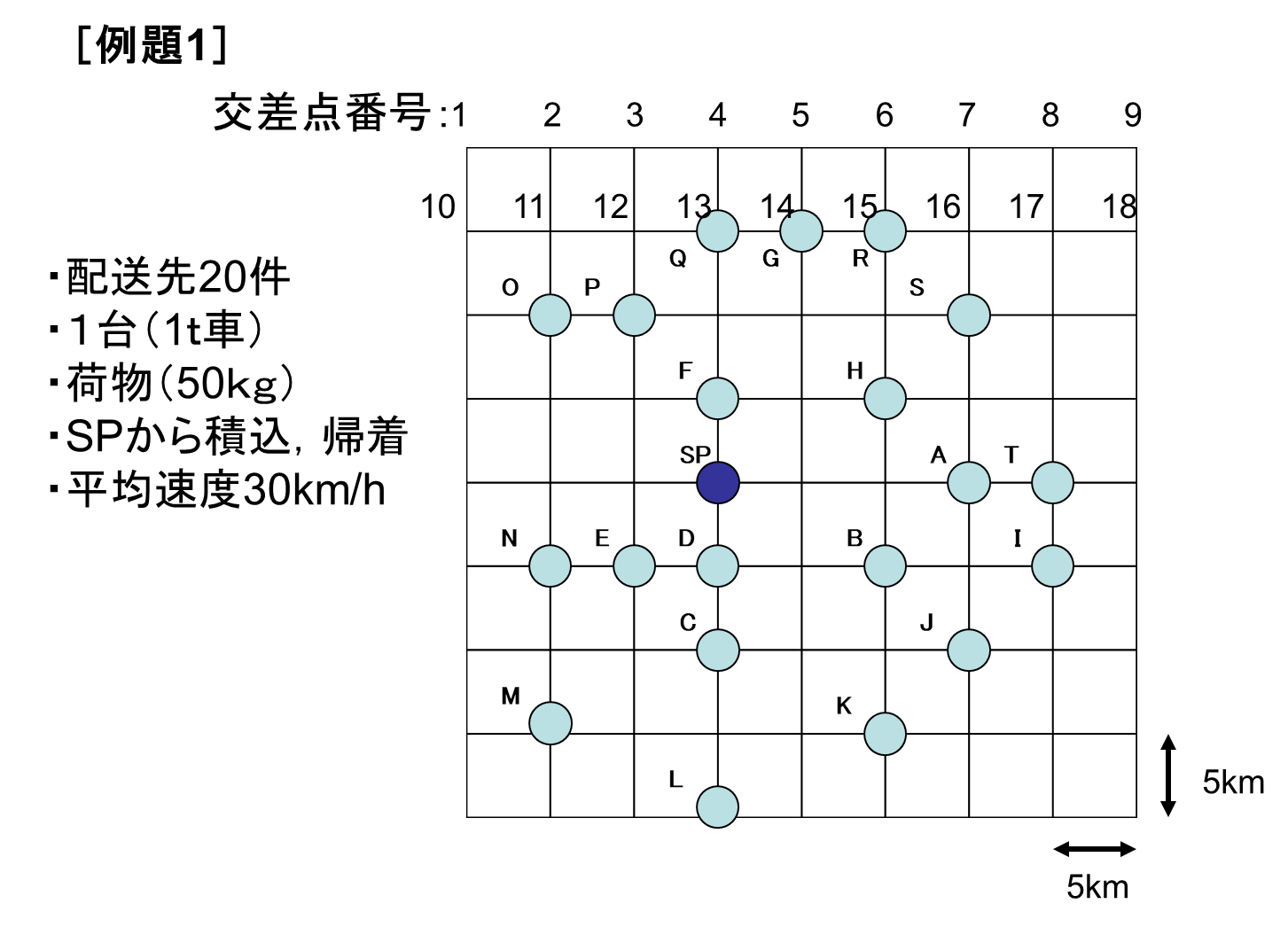
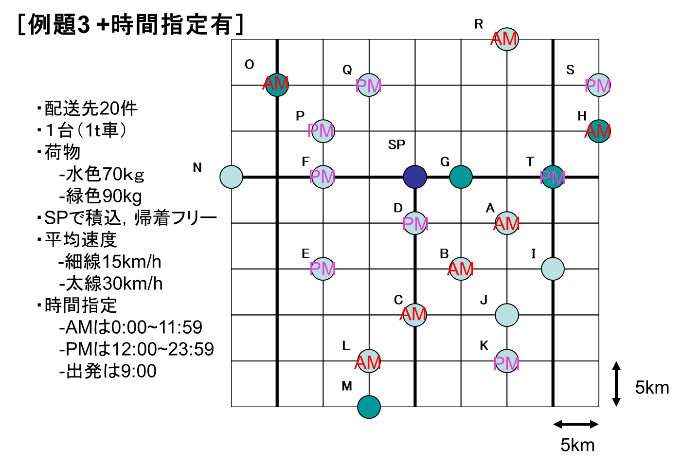
1. 問題の概要

　縦横の道が走った町を考え, その交点を交差点と呼び, 道路上のみを通れるとする.

　特定の地点に開始地点(SP), 配送先が存在し, SPから出発して配送先をすべて回る経路の内,所要時間(コスト)が最小になるような経路を求める.   
　また制約条件として, 帰着,荷重制限,到着時間指定を設定し,経路はこれを満たさなければならないとする.   
　ー帰着: すべての到着地点を回った後,SPに戻る.

　ー荷重制限: 一度に運べる荷重に上限をつけ,SPに積み込みに戻らせる.

　ー到着時間指定: 指定された時間帯にのみ訪問できる.   
 今回は午前/午後の2時間帯を想定した.   
  
問題として,以下の4問を使用した. 4問目は3問目に時間帯指定の条件を付け加えたものである. 時間帯はサイコロを使って決定した.



1. アルゴリズム
   1. 概要・処理の流れ

|  |  |  |
| --- | --- | --- |
|  | 処理 | データ構造 |
| -Step1 | 町の道の情報を計算可能な形に変換する | 町を交差点同士の接続を表した有向グラフの二次元行列で表現する. |
| -Step2 | SP,配送先(以下まとめて目的地)同士の最短到達時間とその経路を求める | 目的地クラスには,名前,地点,荷重,時間指定,各目的地への最短到達時間とその経路を保存する |
| -Step3 | グリーディ法で暫定解を求める | 経路は目的地の配達順を表す一次リストで表現できる |
| -Step4 | 焼きなまし法で改善する |  |

* 1. 町のグラフ化  
     Step1は次の手順からなる.  
     i)町の交差点上に番号を振っていく.以下この番号を"地点"と呼ぶ.  
     ii)隣り合っていて直接移動可能な地点を二次元配列で表す.  
     iii)この二次元配列を外部出力する.(今回はCSV形式を利用する)
  2. 各目的地同士の最短経路の計算(ダイクストラ法)  
     Step2は次の手順からなる.  
     i) 始点を訪問済リストに加え,到達可能な全ての地点とその所要時間,自身の地点を訪問リストに加える.

ii) 訪問リストから所要時間最小の要素を取り出し,未訪問でなければ次の要素を   
 取り出す.

iii) 取り出した地点を未訪問リストから除外し,訪問済みリストに所要時間と親地 点と共に加える. 到達可能な全ての地点と,その到達時間と自身の所要時間を  
 足したものを格納する.

iv) 全ての目的地が訪問済リストに加えられるまでii)iii)を繰り返す.

v) 各目的地について,親ノードを順に辿って始点までの経路を作成し,所要時間と  
 共に格納する.

* 1. グリーディ法による暫定解の作成  
     Step3は次の手順からなる.  
     i) 配達順リストにSPを加える  
     ii) 未訪問の目的地の内,前回配達順リストに加えられた目的地から最も近い　  
     　目的地を配達順リストに追加し,その目的地の荷重を運搬荷重に足す.  
     iii)もし運搬荷重が(往復回数/総荷重)を越えたら,SPを配達順リストに加え,  
     　運搬荷重をリセットする.
  2. 焼きなまし法で暫定解を元に解探索  
     Step4は次の手順からなる.  
     i) ,とする[1]  
     ii) 近傍状態を生成する.もしこの近傍状態が制約条件を満たさなければ生成をやり直す.  
     iii)この近傍状態と現在状態の所要時間(評価値)の差をとし,の場合は確定で,若しくはの確率で現在状態を近傍状態に遷移させる.  
     iv) ii, iiiを繰り返しながら,   
      一定回数(この間を熱平衡と呼ぶ)毎に とする.(徐冷)   
     v)となったら解探索を終了する.

*焼きなまし法(Simulated Annealing, SAと略記)は,大域的最適化問題への汎用の乱択アルゴリズムである。広大な探索空間内の与えられた関数の大域的最適解に対して、よい近似を与える。 S. Kirkpatrick、C. D. Gelatt、M. P. Vecchiらが1983年に考案し、1985年に V. Cerny が再発見した。*

*Wikipedia焼きなまし法 より抜粋*

* 1. 近傍状態の生成  
     　今回の焼きなまし法内での近傍状態は,「互換」「移動」「逆順」の3通りの方法の内1つを確率で選択し実行することで生成しており,複数の生成方法を使うことによってより効率良く捜索を行うことを意図している. (根拠となる実験は5.2にて行っている. )  
     　それぞれの方法について説明する.

・「互換」  
 　ランダムに2つの目的地を選択し,二者を入れ替える.

・「移動」  
 　ランダムに1つの目的地と1つの移動先を選択し,移動させる.

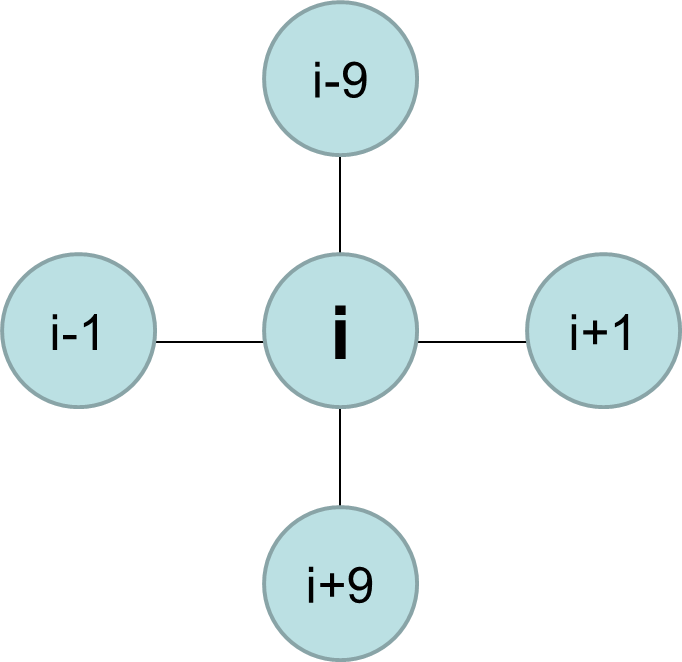
・「逆順」  
 　ランダムに2つの目的地を選択し,二者の間にある目的地の順番をひ 　っくり返す.

* 1. 時間指定有のコスト計算について  
     　到着時間指定の設定があり,かつ現在時刻と一致しなかった場合,その場で指定時間まで待機することとする. これにより,コスト計算以外の部分は3問目までと同様のアルゴリズム・プログラムで解を求めることが出来る.  
     　時間指定と一致しなかった時のコストは跳ね上がることになるが, 暫定解の作成で時間を考慮しなくても良い点, 時間帯の移り際でのルート取りが特に柔軟に行える点などもメリットとして挙げられる.

1. プログラムの解説

作成したプログラムにおいて,重要な部分や工夫した部分などを解説する.

このプログラムの全文は以下のGitHubにて公開している.   
　https://github.com/tanuki12hiromasa/MakeDirGraph (有効グラフの作成,C)  
 https://github.com/tanuki12hiromasa/RoutingProgram2 (最短経路探索,C#)

* 1. 有効グラフ作成部分 (x2graph2.c main() より抜粋)  
     町の情報を基に,有効グラフを生成する部分である.   
     交差点番号は左上から順に振られていて,9行9列の道があるので,右図のように今いる交差点から上下は,左右はの番号が隣接している交差点となる.   
     端や細線・太線は商や余剰の値で判断することが出来る.

int w[81][81];

int i,j;

for(i=0;i<81;i++)

for(j=0;j<81;j++)

w[i][j]=0;

for(i=0;i<81;i++){

if(i>=9){

if(i%9==1||i%9==4||i%9==7) w[i][i-9]=1;

else w[i][i-9]=3; }

if(i<72){

if(i%9==1||i%9==4||i%9==7) w[i][i+9]=1;

else w[i][i+9]=3; }

if(i%9!=0){

if(i>=9&&i<=17||i>=45&&i<=53)w[i][i-1]=1;

else w[i][i-1]=3; }

if(i%9!=8){

if(i>=9&&i<=17||i>=45&&i<=53)w[i][i+1]=1;

else w[i][i+1]=3; }

}

すべての交差点を0で初期化

商で行,余剰で列を判断する

* 1. ダイクストラ法 (PathFinder.cs findRoute() より抜粋)  
     ダイクストラ法を用いて,目的地と目的地の間の最短時間経路を探索する. 経路を保存する際は,行先の目的地から1つずつ交差点を辿って行きながら経路リストに格納し,始点の目的地に到達したら順番をひっくり返している.

//dest[]は目的地データを扱うクラスの配列.   
for(int i = 0; i < dest.Length; i++){

　 int dLeft = dest.Length;

var nextNodeList = new List<Node>(); //訪問する目的地のリスト.

var visitedNodes = new Node[nCross]; //訪問済目的地の場所,訪問元交差点の番地,コストを格納

nextNodeList.Add(new Node { pos = dest[i].pos, prev = -1, cost = 0 });

while (dLeft > 0){

nextNodeList.Sort((a, b) => a.cost - b.cost); //訪問リストをコスト昇順に並び替える

var node = nextNodeList[0];

nextNodeList.RemoveAt(0);

if (visitedNodes[node.pos] == null){

visitedNodes[node.pos] = node;

if (isDest[node.pos] != -1) dLeft--;

for (int n = 0; n < nCross; n++){

if (map[node.pos, n] != 0)

nextNodeList.Add(new Node { pos = n, prev = node.pos,   
 cost = node.cost + map[node.pos,n] })

}

}

}

for(int j = 0; j < dest.Length; j++){ //prevを辿ってrouteに格納

if (i == j) continue;

var pos = dest[j].pos;

while (pos != dest[i].pos){

dest[i].route[j].Add(pos);

pos = visitedNodes[pos].prev;

}

dest[i].route[j].Reverse();

dest[i].cost[j] = visitedNodes[dest[j].pos].cost;

}

}

* 1. 近傍状態生成 (PathFinder.cs makepath()より抜粋)  
       
     　定数numofnodeは,目的地の総数を表している.ここでは,事前に設定した発生確率の比(互換:移動:逆順= changeRatio : insertRatio : reverseRatio)の通りになるように,0以上1未満の乱数の値の範囲で制御している.

//互換,移動,逆順のどれかをランダムに発生させる.

//生成確率の比を 互換:移動:逆順= changeRatio:insertRatio:reverseRatio で表現する

var dice = random.NextDouble();

if (dice > (insertRatio + reverseRatio) / (changeRatio + insertRatio + reverseRatio)){

var fromNum = random.Next(1, numofnode);

var toNum = random.Next(1, numofnode);

int transdest = nextpath[fromNum];

nextpath[fromNum] = nextpath[toNum];

nextpath[toNum] = transdest;

}

else if (dice > reverseRatio / (changeRatio + insertRatio + reverseRatio)){

do{

var fromNum = random.Next(1, numofnode + 1);

var toNum = random.Next(1, numofnode + 1);

int transdest = nextpath[fromNum];

nextpath.RemoveAt(fromNum);

nextpath.Insert(toNum, transdest);

} while (0.4 > random.NextDouble());//40%の確率で再度,移動が行われる

}

else{

var fromNum = random.Next(1, numofnode + 1);

var toNum = random.Next(1, numofnode + 1);

nextpath.Reverse(Math.Min(fromNum, toNum), Math.Abs(fromNum - toNum));

}

2つの目的地を入れ替え

1つの目的地を別の目的地間へ移動

2つの目的地間の並びを逆にする

* 1. 荷重計算 (PathFinder.cs)  
       
     　この関数は,荷重制限が守られているかチェックする際に用いる.   
     配達経路リストの順に進み,SPに到着すると,運搬荷重を0に戻す.  
     リストの最後までチェックしたら,運搬荷重の最大値を返す.

最も大きかった時の荷重を返す

SPに着いたら荷重を0に戻す

int MaxSumWeight(List<int> path)

{

int maxsum = 0;

int sum = 0;

for(int i = 0; i < path.Count; i++)

{

sum += dest[path[i]].weight; //dest[]は目的地データの配列

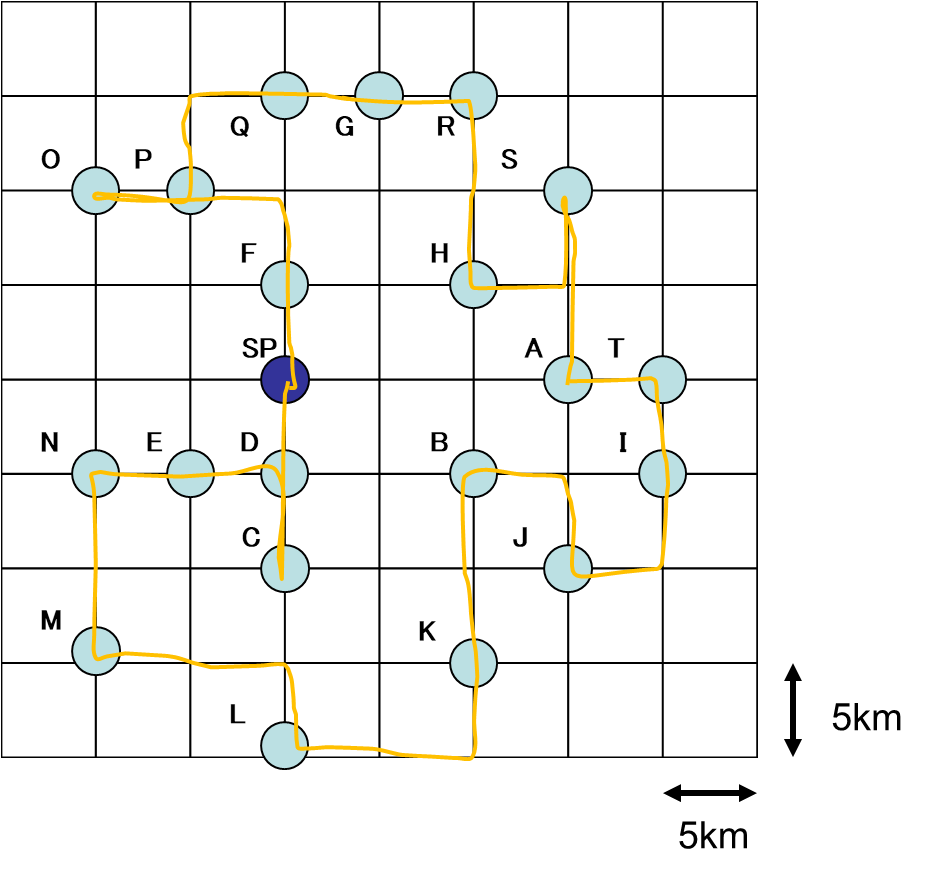
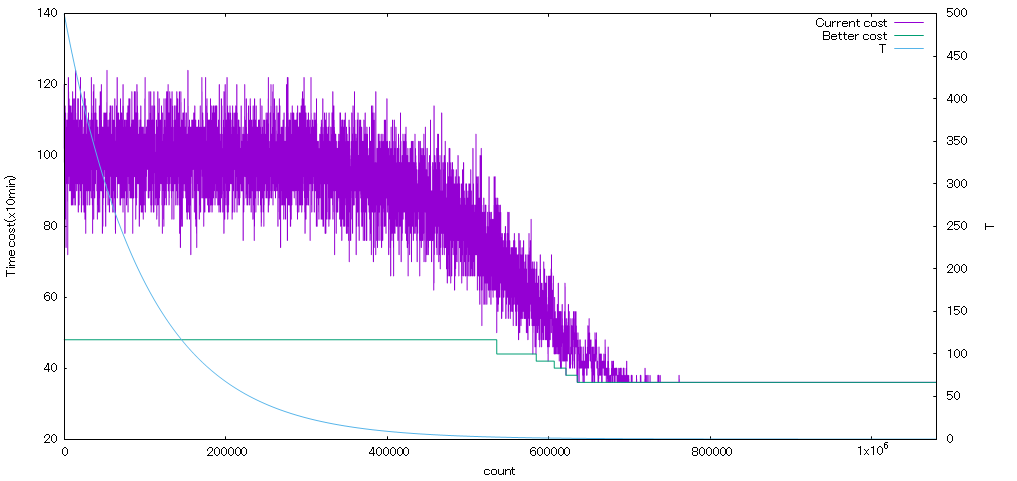
maxsum = Math.Max(maxsum, sum);

if (path[i] == startpoint)

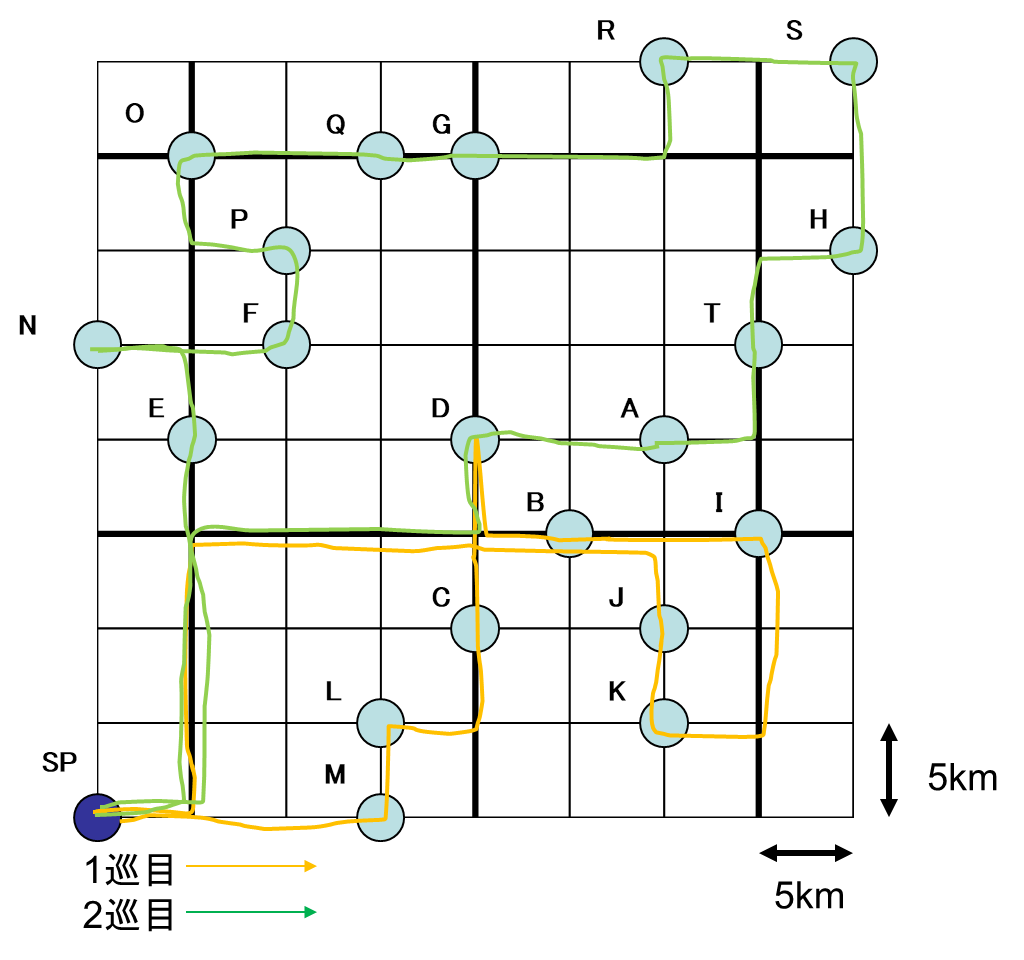
sum = 0;

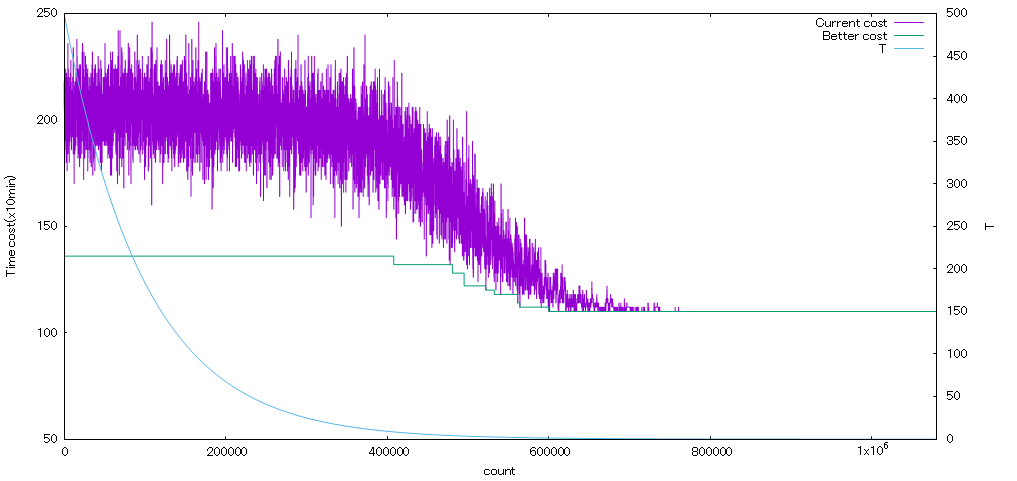
}

return maxsum;  
}

1. 実行結果  
   どの問題についても,  
   , , , 熱平衡状態でのループ数=10,   
   互換:移動:逆順 の実行条件で実行している.
   1. 例題1  
      配送順: SP-C-D-E-N-M-L-K-B-J-I-T-A-S-H-R-G-Q-P-O-F-SP  
      所要時間: 6h 00m  
      暫定解の所要時間: 8h 00m  
        
      
   2. 例題2  
      配送順:SP-M-L-C-D-B-I-K-J-SP-E-N-F-P-O-Q-G-R-S-H-T-A-SP

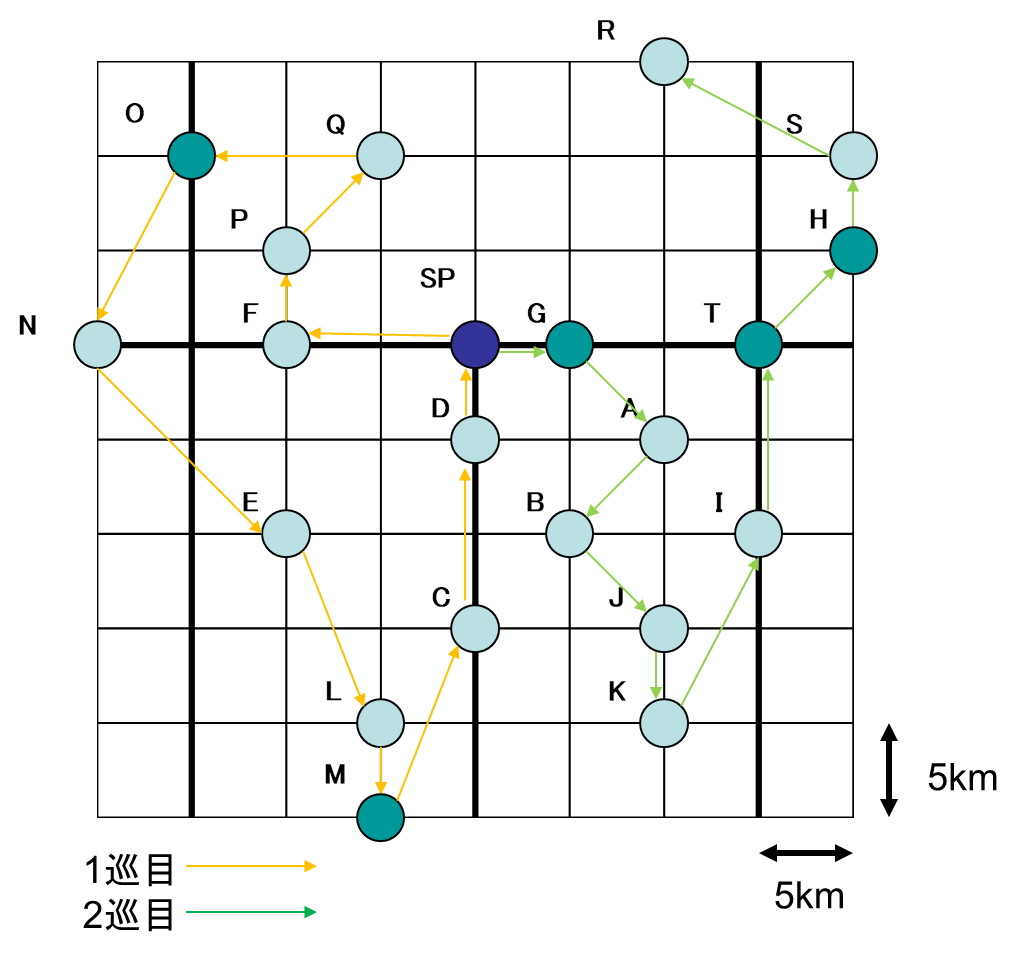
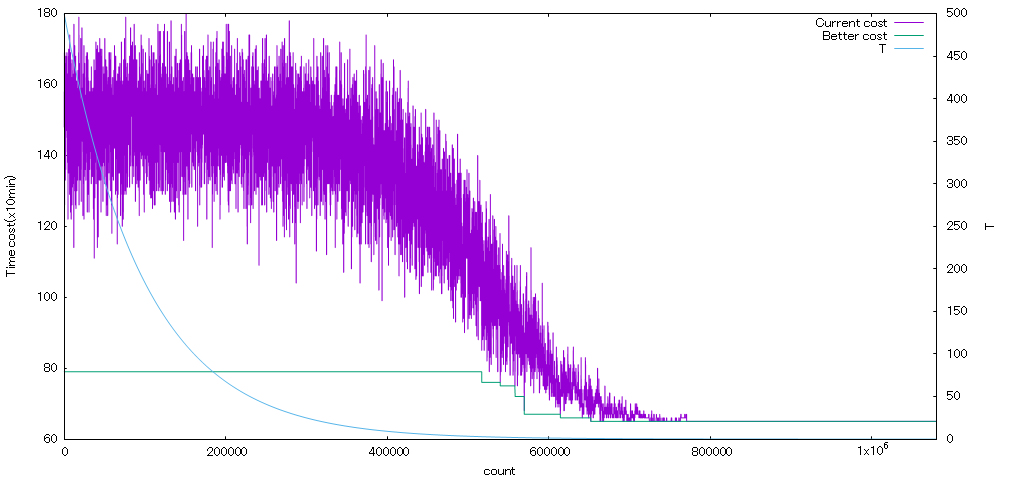
所要時間:18h 20m

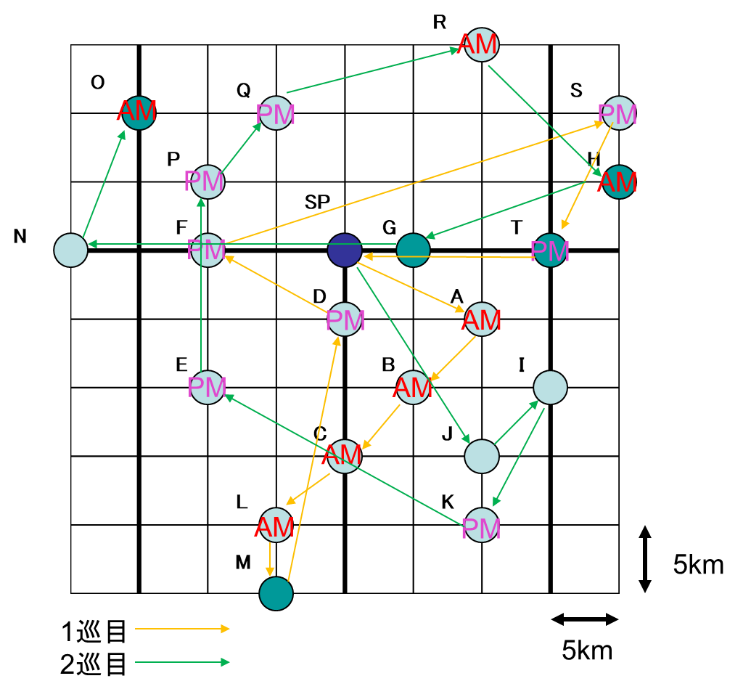
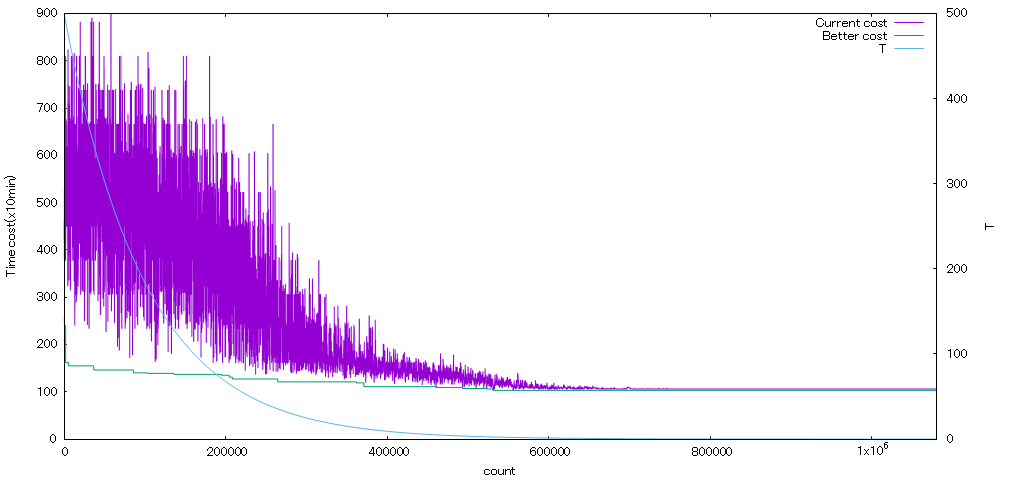
暫定解の所要時間:22h 40m  




* 1. 例題3  
     配送順:SP-F-P-Q-O-N-E-L-M-C-D-SP-G-A-B-J-K-I-T-H-S-R

所要時間:10h 50m

暫定解の所要時間:13h 10m  
  


* 1. 例題3+時間制限  
     配送順: SP-A-B-C-L-M-D-F-S-T-SP-J-I-K-P-Q-R-H-G-N-O  
     所要時間: 17h 10m  
     暫定解の所要時間: 135h 00m  
       
     

1. 考察  
   SA法の温度降下と, 近傍状態の生成について例題3を用いて検証を行った. 各実行条件で5回実行を行い,その内の最良時間と平均時間を記録した.   
   1. 焼きなまし法の温度設定の検証  
      　焼きなまし法において温度管理が重要であることを確かめるため,温度に関する条件を変えて,実行を行った. 条件は以下の3つ.  
      ー E1:をから不変のまま計算を繰り返す  
      ー E2:を極端に大きくしたり小さくしてみる,   
      ー E3:を指数的にではなく線形的に減少させる).  
      対照実験となるように,計算回数が同じになるようやループ数の設定を工夫している. 比較のため,通常のSA法を用いた条件E0 (, , , 近傍状態は「移動」のみとしている)のときの実行結果も併せて示す.

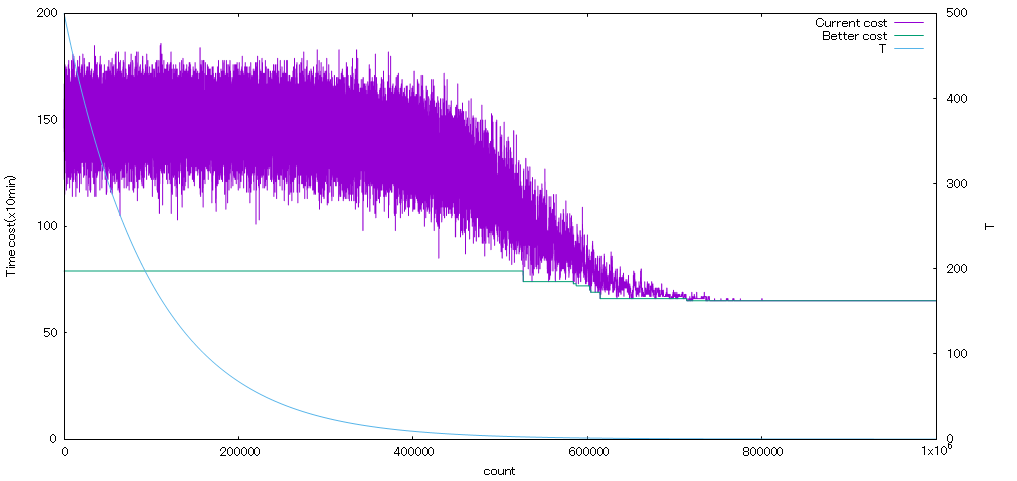
表1　SA法の比較 (例題3)

は共通. 初期暫定解はgreedy法で13h10m. 計算総回数は全て約108万回で揃えている.

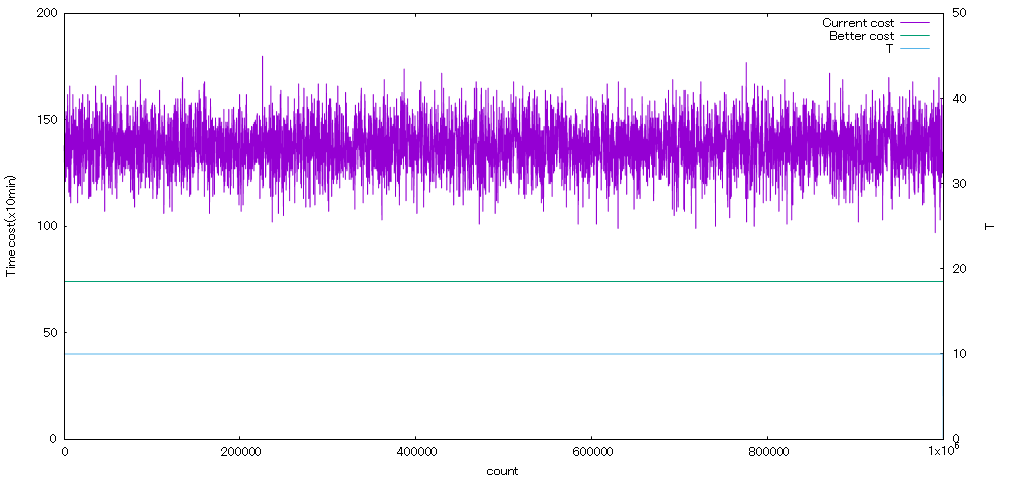
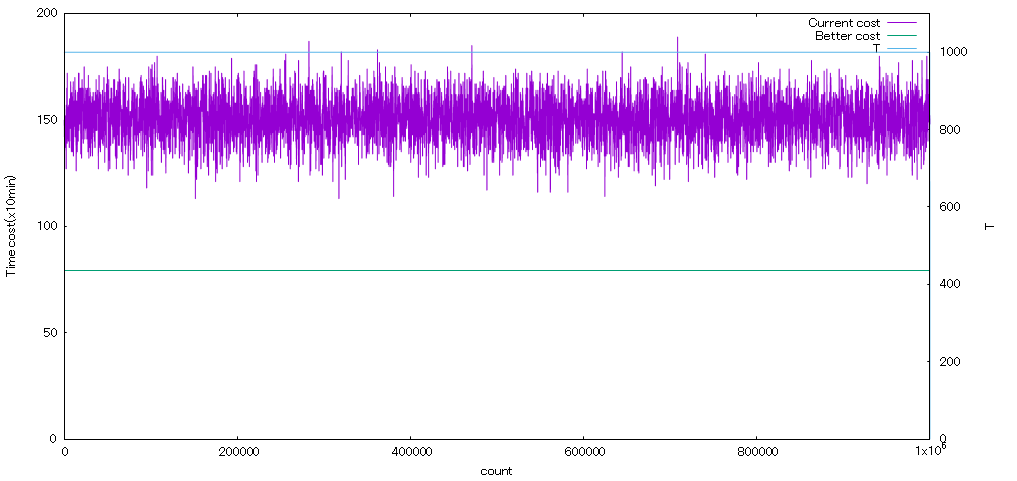
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 実行条件 |  |  | 平衡状態あたりのループ数 | 最良時間  (5回の平均時間) |
| E0通常方式 | 500 | 0.9999 | 10 | 10h 50m(11h 04m) |
| E1 温度一定 | 1000 | 100000 | 1080000 | 13h 10m(13h 10m) |
| 10 | 1000 | 1080000 | 12h 20m(12h 54m) |
| 1 | 100 | 1080000 | 10h 50m(10h 50m) |
| 0.1 | 10 | 1080000 | 11h 30m(11h 30m) |
| E2 T0を変化 |  | 0.99969 | 10 | 10h 50m(11h 06m) |
| 50000 | 0.99986 | 10 | 10h 50m(11h 02m) |
| 5 | 0.99995 | 10 | 10h 50m(11h 02m) |
| 0.5 | 0.99996 | 10 | 11h 00m(11h 16m) |
| E3 を線形減少 | 500 |  | 10 | 11h 10m(11h 34m) |
| 50 |  | 10 | 10h 50m(11h 08m) |
| 5 |  | 10 | 10h 50m(11h 10m) |
| 0.5 |  | 10 | 11h 00m(11h 06m) |

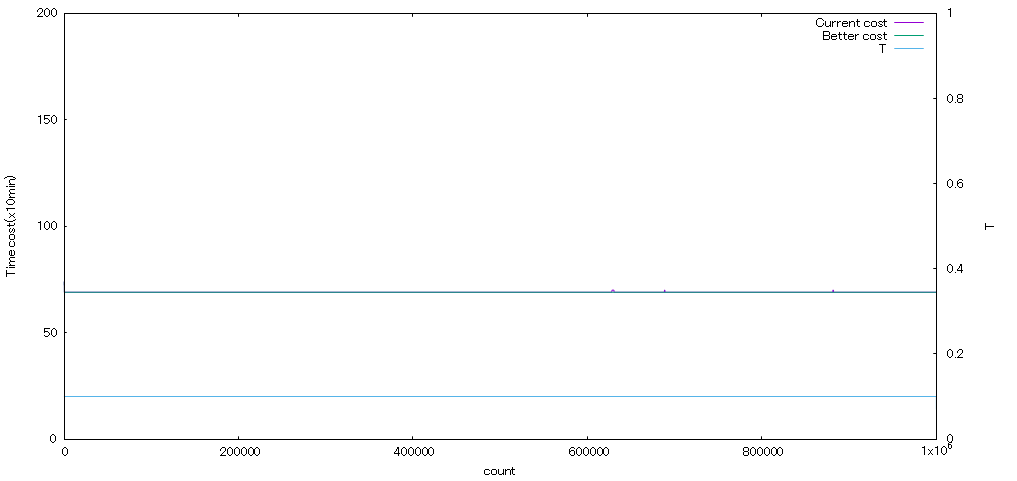
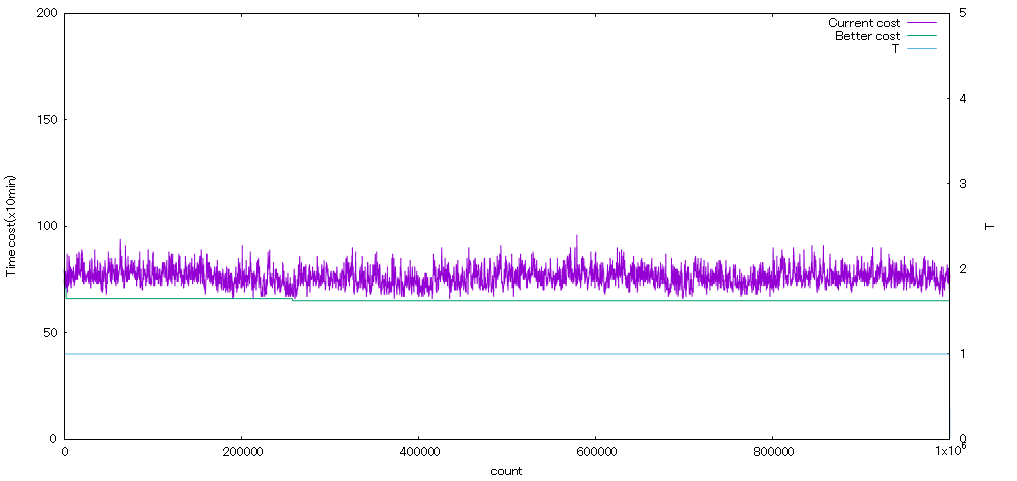
紫線:現在コスト, 緑線:最善コスト, 青線:温度T

横軸:計算回数, 左縦軸:コスト(x10min), 右縦軸:温度



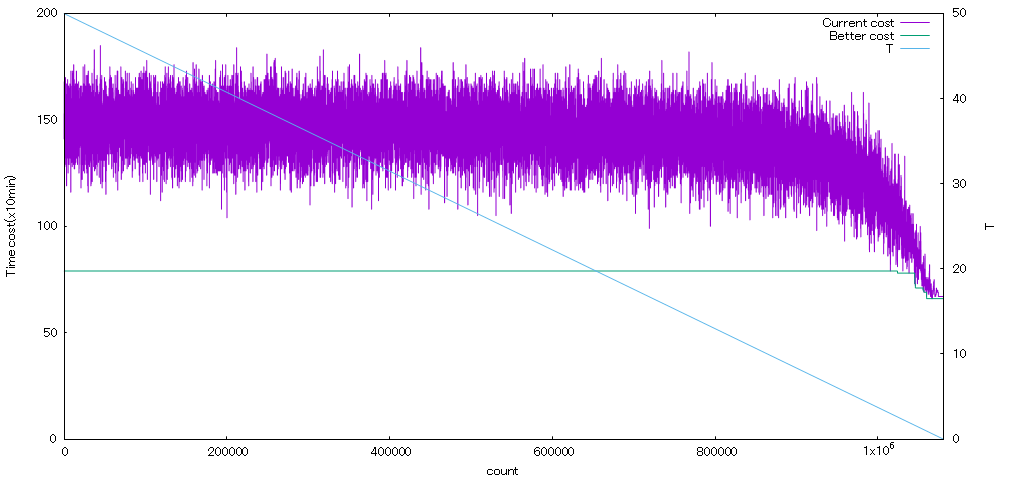
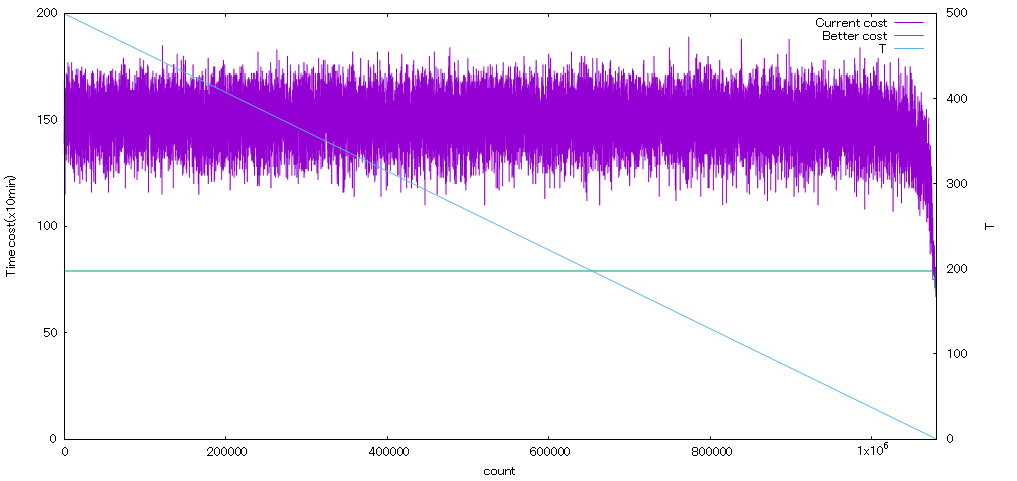
*図5-E0* 標準的なSA法で解いた時のコスト遷移

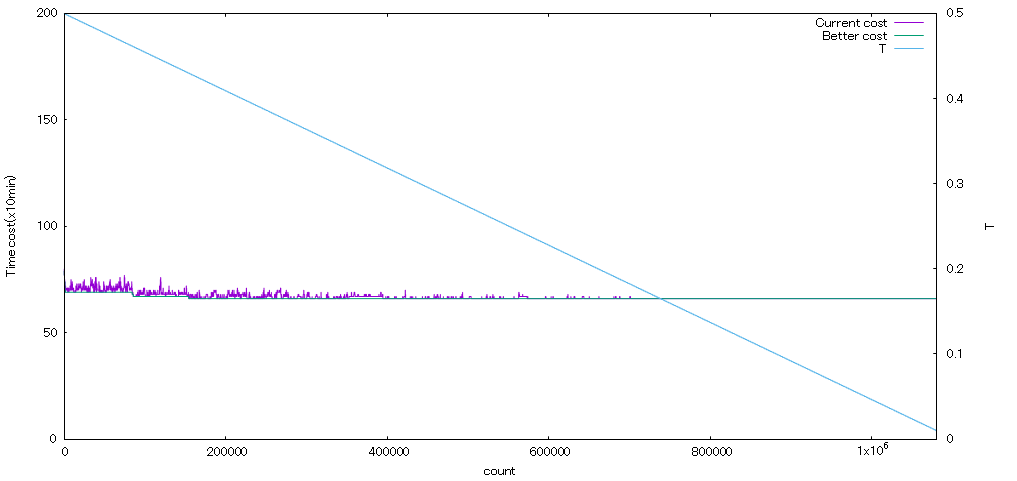
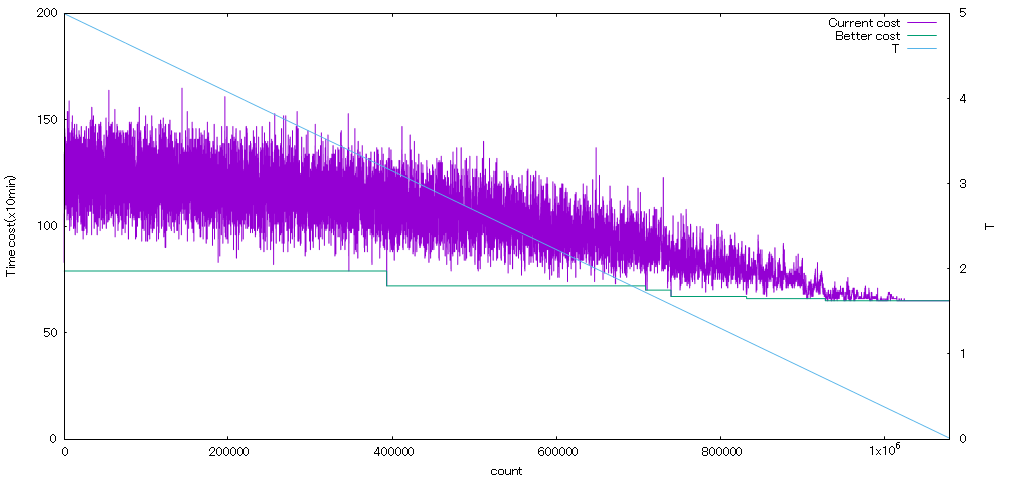




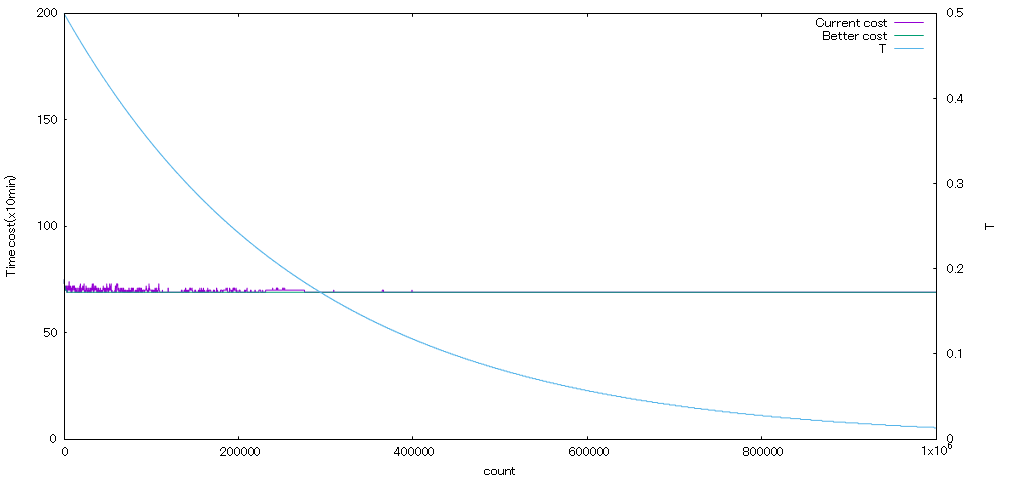
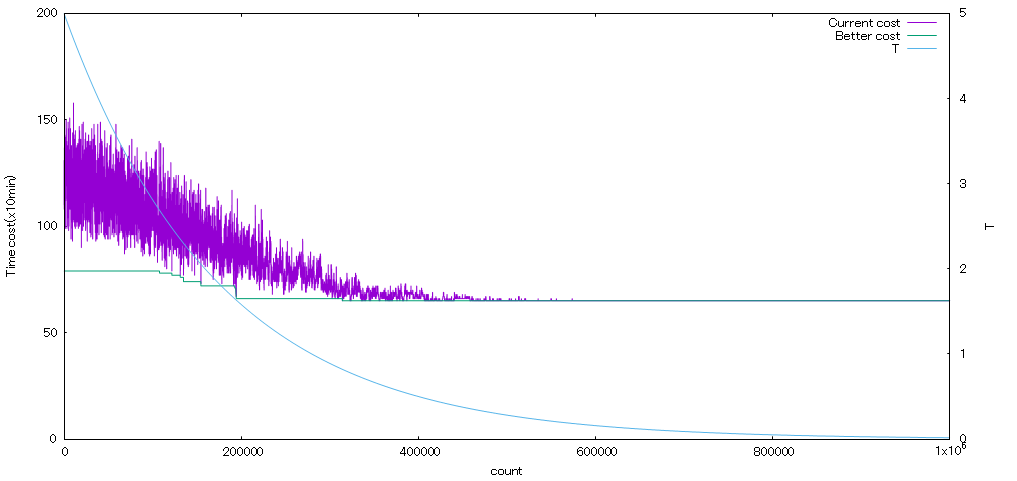
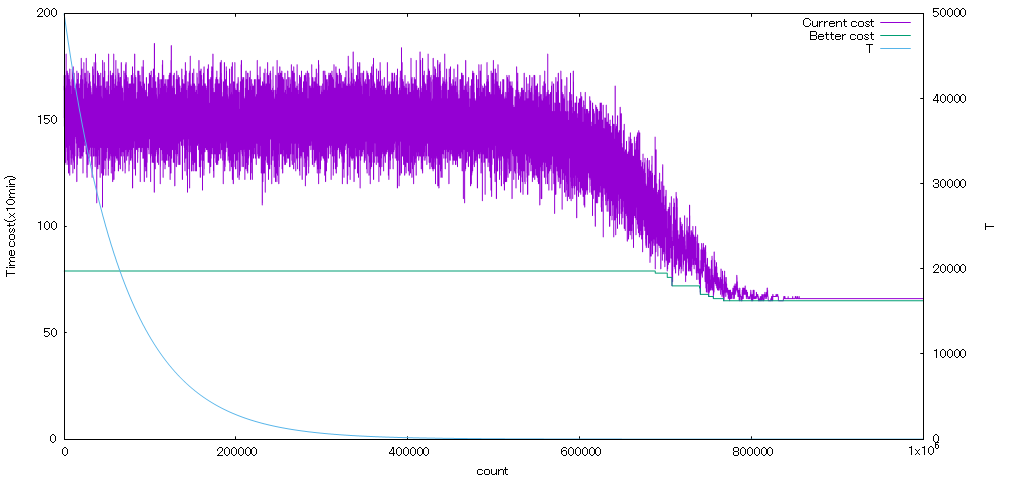
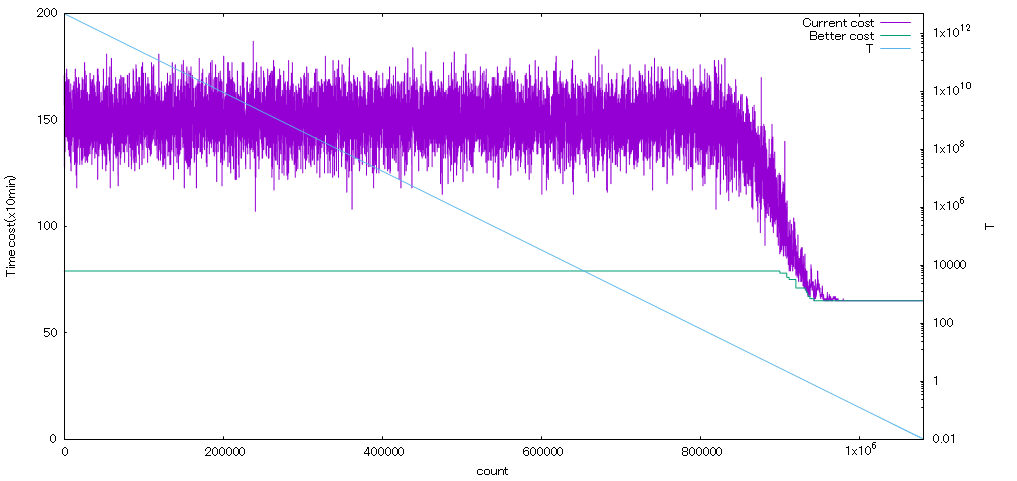
*図5-E1* (定数)としたときのコスト遷移

(右軸は対数)





*図5-E3 を*線形に減少させる方法のコスト遷移.



*図5-E2* 計算回数を変えず,の大きさを変える方法のコスト遷移.

　 E1,E2,E3の実行条件下でのコスト遷移と最良解の結果から考察を行う.

1. E1について. 図5-E1を見ると,初期温度が高いままだと状態は発散したまま収束せず,逆に低すぎると局所最適解から脱出できないことが分かる. T=1のときに良い最良解が出てきているが,T=10とT=0.1の結果から良い最適解の出る範囲が非常に狭いことが分かる.
2. E2について. など小さな初期温度から始めた場合は結果が少し悪くなっている反面,まで極端に大きくした初期温度を用いても,著しい効率の減少は起こっていない.
3. E3について. など初期温度が小さいときはE2の時と似たような結果が出るが,良い結果が出る初期温度の上界が明らかに小さく, で収束しきれない状態になっている. 図5-E3の左上のグラフから,収束が始まってから終わるまでが非常に短くなってしまっていることが分かる.

　図5-E0～図5-E3の現在コストとから,解の収束は凡そ)の範囲で起こっていることが分かる. ここで,状態評価値の偏差について考えると,コストの設定によっては変動し,の解収束範囲もそれに比例して変動する. E2,E3の結果から,を線形に減少させる場合は適正なの範囲を探してSA法を実行する必要があることが分かるが,を指数的に減少させる場合は極端に大きな値から開始しても十分に有用な解を求めることが出来ることがわかる.  
　今回は比較のための値は計算回数が同じになるよう変動させたが,を変動させなくても,と,に対してループ数は高々 のオーダーなので,の増大に対して計算時間の損失は大きくならない. よって,は想定されるの値に対して十分大きな値を設定すればよい.

　以上より,温度降下の式をとするのは,多様な問題設定のスケールにまとめて対応することができる優れた方法であるといえる.

* 1. 近傍状態の生成方法の検証  
     近傍状態の生成比を変えて,結果がどのように変わるか検証した.   
       
      表2　近傍状態の生成方法の比較 (例題3)

|  |  |
| --- | --- |
| 互換 : 移動 : 逆順 | 最良時間  (5回の平均時間) |
| 1 : 0 : 0 | 11h 20m(11h 28m) |
| 0 : 1 : 0 (E0) | 10h 50m(11h 04m) |
| 0 : 0 : 1 | 11h 20m(11h 20m) |
| 1 : 1: 0 | 10h 50m(11h 16m) |
| 1 : 0 : 1 | 11h 20m(11h 20m) |
| 0 : 1 : 1 | 10h 50m(11h 00m) |
| 1 : 1 : 1 | 10h 50m(10h 54m) |
| 2 : 3 : 5 | 10h 50m(10h 56m) |

　まず悪い結果として,「互換」と「逆順」のそれぞれ単体のもの,二者を合わせたものが挙げられる. これらは,2点を選んで入れ替える,という方法をとっているため,到達不可能な状態が存在しており,その中に最良解も含まれてしまっていたと考えられる.   
　良い結果が出たものとしては,3つの方式全てを使用した近傍状態生成(1:1:1)が最も性能が良いので,様々な生成法を複合させるのは少なくともこの問題においては有効であるといえる.   
　三者の生成比については,色々変えて試してみたのだが,良い場合でも2:3:5の例のような結果で,はっきりと更なる改善が見られるような比は見つからなかった. これは,焼きなまし法がその仕組み上,沢山の回数をこなすため,多少の確率変化では結果に違いが表れにくいのではないかと考えられる.

* 1. 4.実行結果 について  
     　各問の実行にあたっては,5章での考察を踏まえた実行条件を用いた.どの問題においても, グリーディ法の解よりも良い結果を得ることができている.  
     　また,4問目は,コスト計算を工夫したことにより,コスト計算関数とデータ読み込み以外でプログラムの変更をせずに計算可能になったことがとても大きい.4.4のグラフからもこの計算方法でSA法が上手く働いていることが分かる.

1. まとめ  
   　本レポートでは,焼きなまし法を中心に,様々な検証を行った.

　この検証の結果として, 焼きなまし法の遷移確率はというそれぞれの問題毎に異なる規模の値をとる変数に対し, 変数を指数関数的に変化させることによって適切に収束が起こるよう工夫されていることや, 近傍状態の生成は,その方式次第で最良解に到達不可能になってしまったり, 実行のパフォーマンスを上昇させたりととても重要な部分であることが分かった.

引用・参考文献  
[1]Kirkpatrick, S.; Gelatt Jr, C. D.; Vecchi, M. P. (1983). “Optimization by Simulated Annealing”. Science 220 (4598): 671–680.

[2]焼きなまし法-wikipedia https://ja.wikipedia.org/wiki/%E7%84%BC%E3%81%8D%E3%81%AA%E3%81%BE%E3%81%97%E6%B3%95