ダイクストラ法を用いた道路混雑度計算の高速化

富田 竜輔(18X4102)　指導教員 五島 洋行

1. はじめに

日常的に起こる交通渋滞は，人々に，そして経済に様々な悪影響を及ぼす.特に長時間運転し，脳・心臓疾患による過労死の多発職種であるトラック運転手にとっては,これは深刻な問題に思われる[1]. また,時間と労力が奪われることによる経済的損失は甚大なものであることは想像に難くない. このように，交通渋滞は悪影響を及ぼすので,渋滞し易い道路を予測するのは重要である.

道路の混みやすさを調査する方法として，実際に観測する方法と推定する方法がある.前者は組織で行われることが多く莫大な費用と労力をかける必要があるが,後者は必ずしも必要ではない. 推定方法に関して，セルオートマトンによる推定やダイクストラ法を用いた推定などがある[2]. ダイクストラ法は最短経路問題を速く解くアルゴリズムであり,かつ現実の様々な場面で使われる[3].

そこで本研究では,1対全ダイクストラ法を用いた道路混雑度を計算するアルゴリズムについて,後に前提条件で紹介する高速化前のアルゴリズムよりも高速な高速化後アルゴリズムを複数提案する.

高速化前のアルゴリズムでは,ダイクストラ法について該当ノードの探索に時間がかかり,またダイクストラ法により求めた最短経路を元に算出する道路混雑度の計算時間も長いが,高速化後のアルゴリズムではどちらも高速化に成功している.

2. 前提条件

2.1 用語の定義

本研究における重要な単語の意味を本節で述べる. 以下２つが重要である.

一対全ダイクストラ法:始点からその他全ての点までの最短距離,最短経路を求めるダイクストラ法.

道路混雑度：0以上の整数値で，値が高い程混みやすいことを示す.全エッジに対して与えられ，最初は全エッジで0である.

2.2 道路ネットワークのデータ表現

本研究では，双方向グラフのデータをプログラムに読み込ませる. データを上から 1 行ずつ読み込んでいくのだが,その一行には2つのノードとそれらを結ぶ 1 つの有向のエッジが含まれ,最終的にエッジの数だけ読み込まれる. 一行のデータの内容を表した図を図 1 とする.

図 1 一行のデータの内容

2.3 高速化前のアルゴリズム

ダイクストラ法について，最短距離が確定したノードやそれに隣接するノードを探索するが,その際前者は必ず全てのノードを調べ,後者は必ず全てのエッジを調べる. また，1対全ダイクストラ法を実行して求めた最短経路を用いて道路混雑度を算出するが,その方法は,始点以外の全ノードから始点へなぞり,エッジを通過したらその道路混雑度に 1 を足すという方法である.

3. 高速化後のアルゴリズム

3.1 アルゴリズム1:ノード探索の改良

始点と最短距離が確定したノードに隣接するノードの探索を高速化している.

高速化前では，探索する際に必ずエッジ全てを,最短距離確定ノードまたは始点に繋がっているかどうか調べている. それに対し，高速化後のアルゴリズムでは,探索する際に必ず最短距離確定ノードまたは始点に繋がっているエッジだけを調べ,繋がっていないエッジは調べない.

3.2 アルゴリズム2:ヒープ木による探索

ヒープ木を使い，最短距離が確定したノードの探索を高速化している.

高速化前では,最短距離確定ノードを探索する度に必ず全ノードを調べる. 一方，改良後では,ヒープ木の根の情報を調べるだけで終わる.

3.3 アルゴリズム3:混雑度算出の改良

道路混雑度の算出を高速化している.

高速化前のアルゴリズムでは,始点以外の全

ノードから始点へなぞる必要があるのに対して，改良後のアルゴリズムでは，最短経路グラフにおいて，次数が 1 である始点以外のノードから，始点へなぞれば良い．

4. 処理時間比較

4.1 実験環境

数値実験の環境を以下の表 1 に示す．

4.2 アルゴリズム毎の処理時間

道路混雑度算出対象のグラフの規模ごとに，高速化前のアルゴリズムと高速化後のアルゴリズムの計算時間を比べる．ただし，秒単位で測り，比較対象の高速化後のアルゴリズムには 3 種類ある．1 つ目が 3.1 で紹介したアルゴリズムであり，これ以降の図でアルゴリズム 1 と表す．2 つ目は 3.1 と 3.2 を組み合わせたアルゴリズムで，これ以降の図でアルゴリズム 1+2 と表す．最後の 3 つ目は 3.1,3.2,3.3 全てを組み合わせたアルゴリズムで，これ以降の図でアルゴリズム 1+2+3 と表す．

ノード数 1923，エッジ数 5016 の小規模グラフにおける，高速化前と高速化後のアルゴリズムの計算時間を表 2 に表示する．ノード数 3444，エッジ数 9052 の中規模グラフにおける，高速化前と高速化後のアルゴリズムの計算時間を表 3 に表示する．ノード数 6493，エッジ数 17902 の大規模グラフとするにおける，高速化前と高速化後のアルゴリズムの計算時間を表 4 に表示する．

表 1 実験環境

CPU	Intel(R) Core(TM) i5-6200U 2.30GHz
OS	Microsoft Windows8.1 Enterprise
Memory	4.0GB
言語	C

表 2 小規模グラフの場合の計算時間(秒)

改良前アルゴリズム	53
アルゴリズム 1	20
アルゴリズム 1+2	2
アルゴリズム 1+2+3	1

表 3 中規模グラフの場合の計算時間(秒)

改良前アルゴリズム	301
アルゴリズム 1	114
アルゴリズム 1+2	6
アルゴリズム 1+2+3	4

表 4 大規模グラフの場合の計算時間(秒)

改良前アルゴリズム	2,068
アルゴリズム 1	748
アルゴリズム 1+2	26
アルゴリズム 1+2+3	15

5. おわりに

表 2,3,4 から，本研究で提案するアルゴリズムは，改良前アルゴリズムと比べて高速であることが推測できる．

また，アルゴリズム{1}と高速化前のアルゴリズムの計算速度の差は，グラフの規模にあまり影響されない可能性があることも推測できる．実際，小規模グラフでは丁度 2.65 倍の差があり，中規模グラフでは約 2.64 倍の差があり，大規模グラフでは約 2.76 倍の差がある．

参考文献

[1] 松本俊，久保智英，井澤修平，池田大樹，高橋正也，甲田茂樹：”トラックドライバーの過労に影響する働き方と休み方の横断的検討”，労働安全衛生研究，vol.13,no.1,p.1,2020

[2] 友枝明保：“「渋滞」を分析する数理モデルとしてのセルオートマンとその応用”，システム/制御/情報，vol.63,no7,p.1,2019

[3] 安井雄一郎，藤沢克樹，笹島啓史，後藤和茂：“大規模最短路問題に対するダイクストラ法の高速化”，日本オペレーションズ・リサーチ和文論文誌，vol.54,pp.1-5,2011

令和 3 年度卒業研究論文

ダイクストラ法を用いた道路混雑度計算の高速化

法政大学　理工学部　経営システム工学科

経営数理工学研究室

18x4102 富田　竜輔

指導教員　五島　洋行　教授

学科名	経営システム工学科	学籍番号	18x4102
申請者氏名		富田竜輔	
指導教員氏名		五島洋行	

論文要旨

論文題目	ダイクストラ法を用いた道路混雑度計算の高速化

本研究では，ダイクストラ法を用いた道路混雑度を計算するアルゴリズムについて，後に前提条件で紹介する高速化前のアルゴリズムよりも高速なアルゴリズムを複数提案していく．

高速化前アルゴリズムは，ノードの探索と，最短経路を元に算出する道路混雑度の算出に時間がかかる．本研究では，それらの問題に対し，配列を有効利用することで対処している．

ノード数 3444,エッジ数 9052 の道路ネットワークの処理時間について，高速化前では約 300 秒かかるが，これから紹介する高速化アルゴリズムを組み合わせた完成版アルゴリズムは約 4 秒で処理できる．つまり同規模のネットワークなら計算時間を 75 分の 1 に減らせるアルゴリズムを本論文で提案するということである．

目次

第一章　はじめに

1.1 研究背景

日常的に起こる交通渋滞は，人々に，そして経済に様々な悪影響を及ぼす．
交通渋滞が起こるということは，それが起きないより運転時間が長引くということでもあるので，運転手の休み時間を奪い，他のことに割く労力も奪うことになると考えられる．

特に日常的に長時間運転するトラック運転手にとっては，これは深刻な問題に思われる．トラック運転手は脳・心臓疾患による過労死の多発職種で，その職に就いている一部の人々は一か月あたり 100 時間の時間外労働を強いられている[1]．そのような人々にとって，交通渋滞による労働時間の増大はなるべく避けなければならないと思われる．

また，時間と労力が奪われることによる経済的損失は甚大なものであることは想像に難くない．このように，交通渋滞は悪影響を及ぼすので，渋滞し易い道路を予測するのは重要であると思われる．

道路の混みやすさを調査する方法として，実際に観測する方法と推定する方法がある．前者の具体的な観測方法は，トラフィックカウンターを用いる方法と人手による方法に大別される．端的に言えば，機械による方法と人手による方法があるということである．この方法の長所は，正確に道路の混雑度を算出できることである．ただ，どちらの手法でも労力と費用がかかるという短所もある．機械を用いた手法では，購入費を支払わなければならず，またそれを設置する労力が要る．人手による手法では，人件費と測定する労力が要る．

後者の具体的な推定方法は，セルオートマトンによる推定やダイクストラ法を用いた推定などがある[2]．この方法の長所は，道路ネットワークデータさえあれば推定でき，費用がかかりにくいことであるが，推定であるが故に算出される道路の混雑度が不正確であるという短所も存在する．

実際に道路の混み具合を求める際は，複数の手法が用いられることがある．国土交通省が概ね 5 年に一度行っている全国道路・街路交通情勢調査の交通量調査では，トラフィックカウンターと人手により観測する方法と推定する手法の 3 種類が用いられる[3][4]．また，多くの人に利用されている高速道路においても，日常的に NEXCO 東日本がトラフィックカウンターと人手により観測している[5]．

このように，道路の混雑具合を求めるための手法が多くあり，実際にその手法を採用して混み具合を求めていることからも，道路混雑度算出が重要視されていることが分かる．

1.2 研究目的

研究の目的は，道路の渋滞のし易さを道路混雑度として数値化するが，その算出時間を短くすることである．

本研究では，数多ある道路の混み具合の算出法の内ダイクストラ法を用いた推定のみを使い算出し，渋滞のし易さを道路混雑度という 0 以上の整数値で表す．その値が大きい程混雑しやすいことを示す．この道路混雑度は，ダイクストラ法の，始点からその他全ての点への最短経路を用いて算出される．

つまり，ダイクストラ法の処理とそれにより分かる最短経路を使った道路混雑度の算出を実行するプログラムを，処理時間短縮の観点で改良することを目的としている．

ここで，ダイクストラ法のみを用いた推定を行う理由を記述する．混みやすさを観測する方法と推定する方法があると前節で記述したが，前者は私個人の研究では到底できないため推定のみを採用することにした．沢山の推定法の中からダイクストラ法の推定を採用した理由は，最短経路問題を速く解くアルゴリズムであり，かつ現実の様々な場面で使われるからである[6]．本研究では，ダイクストラ法自体の高速化と道路混雑度算出の高速化の両方に取り組んでいるが，実用性のあるダイクストラ法の高速化に取り組んでいる時点で有意義であり，また前節から分かる通り現実で行われている道路混雑度の推定をも行うのでより有意義な研究となる．

第二章　関連知識

本章では，本研究で用いるダイクストラ法のアルゴリズムと，それに関連する知識について記述する．

2.1 ダイクストラ法のアルゴリズム

ノードとエッジで構成されるグラフに適用できる，最短経路問題を効率よく解くアルゴリズムとして知られている．本研究と深く関わるので，このアルゴリズムの処理手順を以下に記述する[7].

図 2.1 から 2.3 で，ノード s,v_1 ，v_2，v_3，v_4,t の横についている丸かっこ内の値は，その時点での暫定的な最短距離を表す．また，以下で登場する赤矢印のポインタは，暫定ないし確定した最短経路を構成している．

ステップ 1

$\mathrm{L} = \emptyset, \delta(\mathrm{s}) = 0, \delta(\mathrm{v}) = \infty \ (\mathrm{v} \neq \mathrm{s})$と変数を初期化する．ここで，L は最短距離が確定したノードの集合を，v は任意のノードを，s は始点を，$\delta(\mathrm{v})$は L に入っている頂点だけを使って v に至る最短距離を表す．図 2.1 を参照.

ステップ 2

頂点 s を L に入れる．次に，s に隣接している各頂点 v について，箇条書きで書かれた以下 2 つを行う．図 2.2 を参照.

・$\delta(\mathrm{v}) = w(s, v)$とする．$w(s, v)$は sv 間のエッジの重みを表す.

・v はポインタで s を指す.

ステップ 3

まだ L に入っていない頂点の中でδの値が最小のものを v とし，v を L に入れる．v の候補が複数ある場合は，任意に一つ選ぶ．次に，v に隣接している頂点の内，まだ L に入っていない頂点 u に対して，箇条書きで書かれた以下 2 つを上から順に行う．図 2.3 を参照．後は，図 2.1,から 2.3 に描かれた点 t が L に入るまでステップ 3 をくり返す.

・$\delta(\mathrm{v})$の新しい値を，$\delta(\mathrm{v}) = \min\{\delta(\mathrm{u}), \delta(\mathrm{u}) + w(v, u)\}$とする．$\min\{\mathrm{a}, \mathrm{b}\}$は a,b の小さい方の値を意味する.

・$\delta(\mathrm{u}) > \delta(\mathrm{u}) + w(v, u)$の場合，u のポインタを v に向ける.

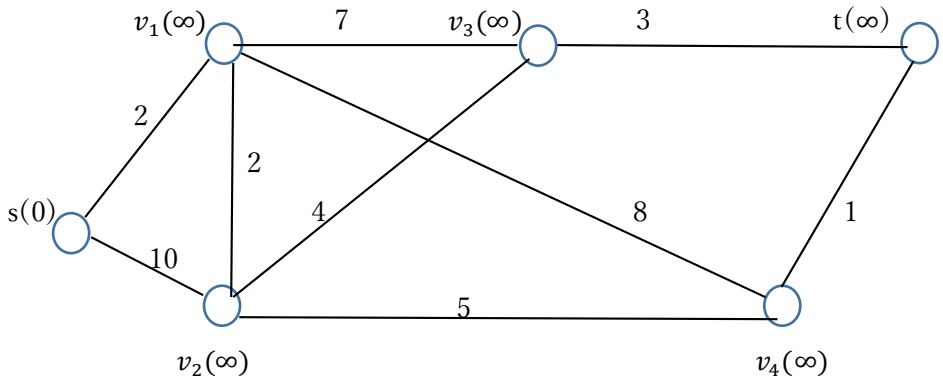

図 2.1 ステップ 1 の例

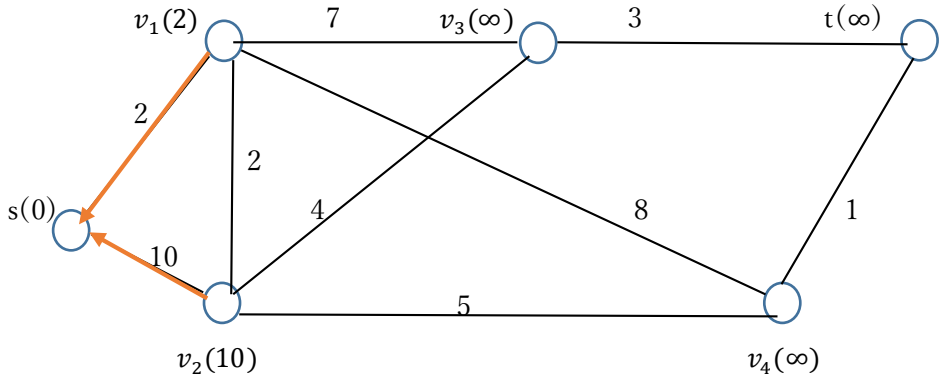

図 2.2 ステップ 2 の例

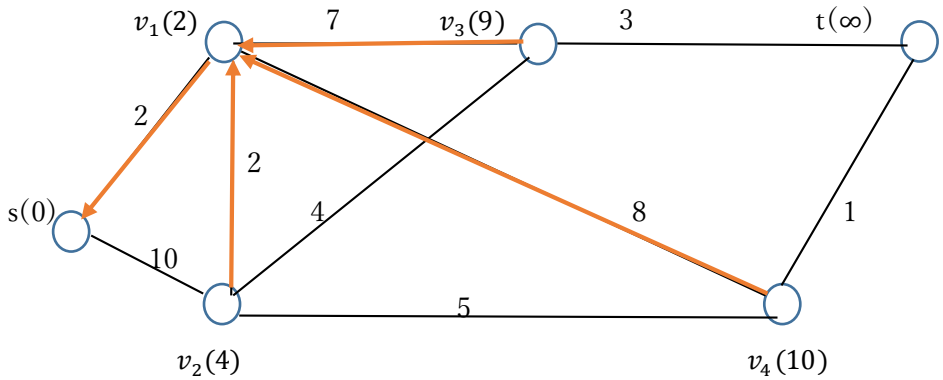

図 2.3 ステップ 3 の例

2.2 ダイクストラ法の特徴

非負の枝長を持つグラフに対する効率的なアルゴリズムとしてダイクストラ法が有名であるが，アルゴリズム中の“次探索点の選択”がボトルネックとなることが知られている．また，動的計画法の特性から，データアクセスパターンの中長期的な予測が困難であり，データの再利用率が低い[6].

2.3 ダイクストラ法と優先キュー

ダイクストラ法の”次探索点の選択”の時間を短縮するために，優先キューというデータ構造が使われてきた．最短経路問題に対する優先キューは，ヒープを用いた優先キューとバケットを用いた優先キューの 2 つに大別できる．前者では，優先度の大小関係により木構造を構成するため，扱うデータの範囲によって性能が左右されることは少なく，実行時間やメモリ要求量が安定的である．後者では，優先度の値に対して予め決定した規則に従いデータを配置させるため，実行時間やメモリ要求量が扱う値の範囲に依存してしまう[6].

2.4 ベルマン-フォード法

ダイクストラ法だけでなく，ベルマン-フォード法も短経路問題を解くアルゴリズムとして有名である．2 点間の最短経路を求めることができる．ダイクストラ法はエッジの重みが非負でなければならないが，ベルマン-フォード法は負であっても正しく動作する．

2.5 ワーシャル-フロイド法

上述した 2 つのアルゴリズムと共に，ワーシャル-フロイド法も最短経路問題を解くアルゴリズムとして有名である．ベルマン-フォード法と同様に，エッジの重みが負であっても正しく問題を解くことができ，負閉路があるならそれを検出することができる．

第三章　前提条件

本章では，次章以降の説明をより理解するために必要な事柄を記述する．

3.1 変数・定数の定義

本節では，以降に登場する変数と定数と言葉を説明する．定義は以下の通りである．なお，文章だけでは分かり辛い説明もある．そのような変数・定数は，適宜図示などして理解できるよう努める．

双方向グラフ ： 道路ネットワークを表現した有向グラフ．2 つのエッジは向いている方向が 180 度逆で，2 つのノードが両端にある．

node_cnt ： 双方向グラフの全ノード数.

edge_cnt ： 双方向グラフの全エッジ数.

v_i $(i = 1,2,3, \cdots, \text{node\_cnt})$ ： 双方向グラフのある 1 つのノード.

e_i $(i = 1,2,3, \cdots, \text{edge\_cnt})$ ： 双方向グラフのある 1 つのエッジ.

V ： v_iだけを要素に持つノードの集合

E ： e_iだけを要素に持つエッジの集合

tail_[e_i] ： エッジe_iの根元のノード．Vの要素.

head_[e_i] ： エッジe_iの指す先のノード．Vの要素.

w_[e_i] ： e_iの重み．端から端へ移動するのに費やすコスト.

READ(V,E,w_[e∈ E]) ： 上記 V,E,w_[e]の情報をプログラムが読み込む.

distance[v ∈ V] ： 始点から v までの，暫定または実際の最短距離.

status[v ∈ V] : v の最短距離が未確定なら-1 で，確定なら 1 である.

flag[v∈ V] : v について distance[v]を更新したことがあるなら 1,ないなら 0 が入る.

father[v∈ V] : (暫定)最短経路における，v の直前のノード.

node[v∈ V] : int 型の num と double*型の dist を持つ Node 型の配列．num は v の値を表し，dist は distance[v]のアドレスを指す．最短距離未確定ノードが入る.

[ORDER BY tail_[e∈ E] ,ASC] : tail_[e]が昇順になるように READ(V,E,w_[e∈ E])を並べかえる.

correspT ： v ∈ Vと，tail_[e∈ E]=v を満たす最大の e の表

correspT[v] ： v ∈ V について，tail_[e∈ E]=v を満たす e の最大値

[correspT[v-1,v],v-1・v ∈ V] : correspT[v-1]+1,correspT[v-1]+2,・・,correspT[v]

HeapSort ： 配列 node に入っているノードを，*dist が親ノード≦子ノードとなるようヒープソートする.

DownHeapsort： ヒープ木の根の深さを，親≤子となるように深くする．

UpHeapsort(x)： ヒープ木のノード x の深さを，親≤子となるように浅くする．

Dijkstra(v ∈ V)：始点を v として，本章 3 節で解説する一対全ダイクストラ法を実行．

s：上記ダイクストラ法の始点を表す．

descendant[v ∈ V]：上記ダイクストラ法で求めた最短経路について，v が子ノードを持つなら 1 として，持たないなら 0 とする．
値が 2 になる時があるが，TF(decendant, v)で説明している．

TF(v ∈ V)：最短経路上の v から s までの全エッジに，道路混雑度として 1 を足す．

TF(decendant, v)：v から s へ，s までの最短経路上を移動し，エッジを通過し終えた際，そのエッジに道路混雑度としての値を足す．最後に v から s までの全ノード all について，decendant[all]=2 とする．どのような値を足すかというと，移動経路上のノードを n として，s から移動して初めて行き着く decendant[n]=2 となる n を N とするなら，N を通過する前は，エッジに通過エッジ数を足す．N を通過した後は，通過前に足した最大値をエッジに足す．最後に v から s までの全ノード all について，decendant[all]=2 とする．

3.2 道路ネットワークのデータ表現

本研究では，双方向グラフのデータをプログラムに読み込ませるが，そのデータについて，本節で詳しく記述する．データを上から 1 行ずつ，上からe_1，e_2，e_3・・,$e_{\mathrm{edge_cnt}}$として読み込んでいくのだが，左から順にtail_[e_i]，head_[e_i]，w_[e_i]が行内にある．つまり一行ごとに以下の図 3.1 のような情報が格納されているということである．全ての行を読み込むと，例えば，図 3.2 のようになる．提案アルゴリズムと深く関わるので，上記のようなデータを読み込み道路混雑度を算出することを頭に入れていただきたい．

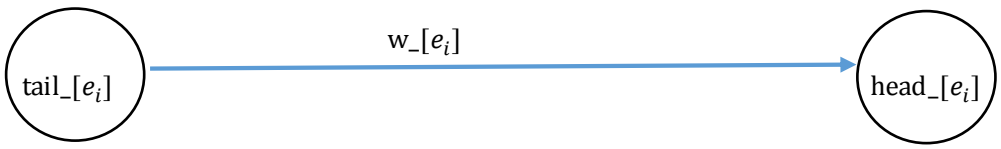

図 3.1 一行のデータの意味

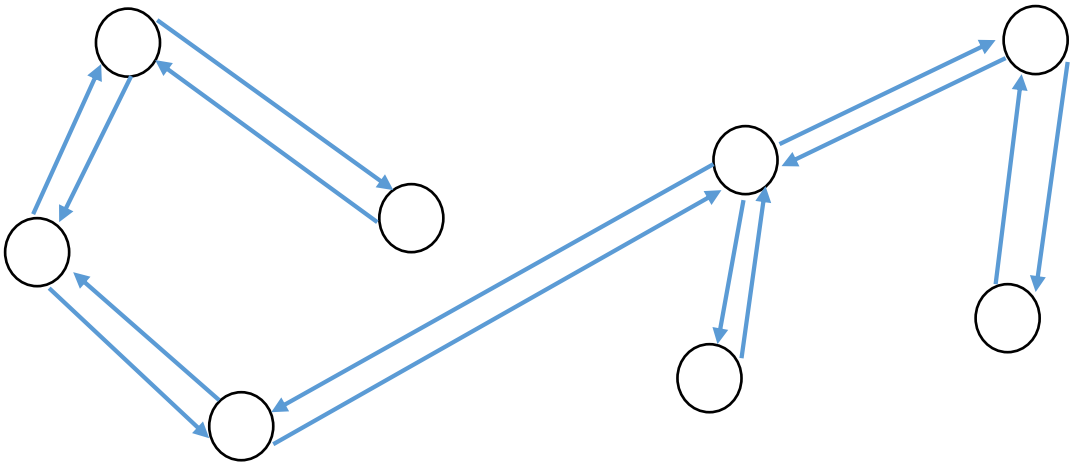

図 3.2 読み込まれる双方向グラフの例

3.3 一対全最短路問題と道路混雑度

ダイクストラ法を用いた最短路問題として，以下 2 つが有名である．2.1 で説明したのは一対一最短路問題である．

始点を変えながら一対全最短路問題を解き，その最短経路データだけをもとに道路混雑度を算出していくと考えて差し支えない．つまり，本研究での道路混雑度は，人間の判断や交通事故などに一切影響されないということである．

・一対一最短路問題：始点と終点を 1 つずつ設定し，始点から終点までの最短経路と最短距離を求める問題.
・一対全最短路問題：始点を 1 つ設定し，始点からその他各点までの最短経路と最短距離を求める問題.

3.4 高速化前のアルゴリズム

後の章での提案アルゴリズムの説明のし易さのため，本節で高速化前のアルゴリズムを記述する．アルゴリズム全体の挙動を Algorithm1 で，Algorithm1 で出てくる一対全最短路問題を解く「Dijkstra」の挙動を Algorithm2 で紹介する．

Algorithm1 道路混雑度計算アルゴリズム

1: READ(V,E,w_[e])
2: **for all** v ∈ V **do**

```
3:     Execute Dijkstra(v)
4:     for all v′ ∈ V do
5:       TF(v′)
6:     end for
7:   end for
```

Algorithm2 一対全ダイクストラ法 / Dijkstra(s)

```
 1:   for all v ∈ V do
 2:     status[v]=-1
 3:     distance[v]=∞
 4:   end for
 5:   status[s]=1
 6:   distance[s]=0
 7:   for all e ∈ E do
 8:     if(tail_[e]  != s){continue}
 9:     x=head_[e]
10:     distance[x]=w[e]
11:   end for
12:   while(1)
13:   dist_min = ∞
14:     u=-1
15:     for all v ∈ V do
16:       if(status[v]==1){continue}
17:       if(dist_min > distance[v]){dist_min= distance[v]}
18:       u=v
19:     end for
20:     if(u==-1){break}
21:     status[u]=1
22:     for all e ∈ E do
23:       if(tail_[e] != u){continue}
24:       x=head_[e]
25:       if(distance[x] > distance[u]+w[e]){distance[x]=distance[u]+w[e]}
26:     end for
27:   end while
```

第 4 章　高速化アルゴリズム 1

本章では，ダイクストラ法実行中での，最短距離が確定したノードに隣接するノードを速く見つけるアルゴリズムについて説明していく．

4.1 アルゴリズム紹介

第 3 章の Algorithm1 の改良版を Algorithm3 とし，Algorithm2 の改良版を Algorithm4 とする．

Algorithm3 隣接ノード探索の高速化

```
1:  READ(V,E,w_[e])
2:  ORDER BY tail_[e],ASC
3:  CREATE correspT
4:  for all v ∈ V do
5:    Execute Dijkstra(v, correspT)
6:    for all v′ ∈ V do
7:      TF(v′)
8:    end for
9:  end for
```

Algorithm4 隣接ノード探索の高速化 / Dijkstra(s,correspT)

```
 1:  for all v ∈ V do
 2:      distance[v]=∞
 3:        status[v]=-1
 4:  end for
 5:  status[s]=1
 6:  distance[s]=0
 7:    correspT[s-1,s].each do |cT|
 8:      x=head_[cT]
 9:      distance[x]=w[cT]
10:    end each
11:  while(1)
12:    dist_min = ∞
```

```
13:     u=-1
14:     for all v ∈ V do
15:       if(status[v]==1){continue}
16:       if(dist_min > distance[v]){dist_min= distance[v]}
17:       u=v
18:     end for
19:     if(u==-1){break}
20:     status[u]=1
21:     correspT[u-1,u].each do |cT|
22:       x=head_[cT]
23:       if(distance[x] > distance[u]+w[cT]){distance[x]=distance[u]+w[cT]}
24:     end each
25:   end while
```

4.2 アルゴリズム挙動例

本節では，本章で取り上げているアルゴリズムの挙動を説明する．分かりやすさのために，動作対象の具体的なグラフを用意する．図 4.1 である．また，その図の元となる，READ(V,E,w_[e])から読み込まれるデータを図 4.2 として示す．また，理解を深めるためここで 3.2 を再読していただきたい．

まず，改良前のアルゴリズムの挙動を具体的に説明する．今回は，[1,2,3,4]の順で始点にする．高速化前と後で処理が変わらない道路混雑度算出の説明は省略する．

status[1]=1,distance[1]=0 とし，他のノードでは順に-1,∞とする．次に tail_[e]=1 となる e を探し，distance[head_[e]]を更新する．必ず for 文を edge_cnt すなわち 8 回実行す

tail_[e]	head_[e]	w_[e]
4	1	4
3	4	2
2	3	7
1	2	8
4	3	1
2	1	6
1	4	5
3	2	3

図 4.2 (V,E,w_[e])

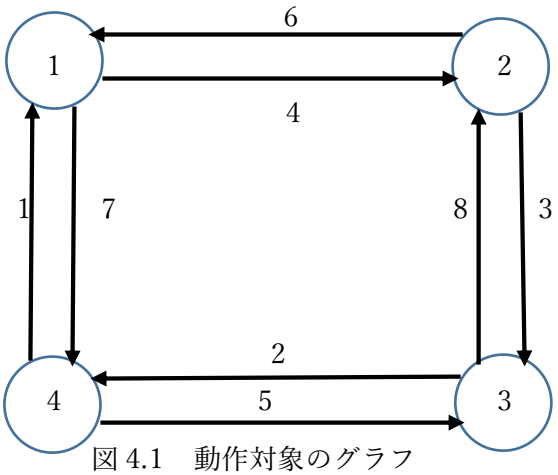

図 4.1　動作対象のグラフ

る．こうして，tail_[e]=1 になる e は 4 と 7 であることが分かり，distance[head_[4]]すなわち distance[2]に w[4]の 8 を，distance[head_[7]]すなわち distance[4]に w[7]の 5 を代入する．次に，status[v]が 1 でないノード v すなわちノード 2,3,4 について，distance[2], distance[3],distance[4]の中で最小の値となるノード v を最短距離が確定しているノードとして扱い，status[v]=1 とするが，最小値が∞の場合ダイクストラ法を終了させる．この場合，distance[4]すなわち 5 が最小となり，status[4]=1 となる．次に，tail_[e]=4 となる e を，for 文を edge_cnt つまり 8 回実行し探し，distance の値を更新する．この例では，該当するエッジは 1 と 5 で，distance[head_[5]]すなわち distance[3]の値を 5+1=6 に更新する．

後は while(1)を break で抜け出すまで同様に実行し，求まった最短経路を元に道路混雑度をはじき出す．また，始点ノード 2,3,4 でも同様にダイクストラ法を実行し道路混雑度を算出すれば良い．

これで高速化前のアルゴリズムは終了したので，以下高速化後のアルゴリズムの挙動を丁寧に説明する．ここでも始点の設定の順番は 1,2,3,4 ということにする．また先程説明したが，改良の前と後では道路混雑度算出の処理に全く違いがないので，その詳しい動作は説明しない．

ここから高速化後の処理を説明していく．まず図 4.2 の配列を読み込む．その直後にそれを tail_[e]の値が昇順になるように，行単位で並び替える．ただし，1 行目からe_1，e_2，・・，$e_{\mathrm{edge_cnt}}$として扱うのは変わらない．次にこの並び替えた配列を元に correspT という配列を作成する．並び替えた後の配列とそのグラフの図と correspT という配列を順に図 4.3,図 4.4,図 4.5 として以下に表示する．ここで，correspT の一行目で，ノード番号とエッジ番号の値が両方 0 であるが，これは番号が 0 のノードやエッジが存在するということを表していない．アルゴリズムが正しく動作するために必要であるため付け加えたのである．

tail_[e]	head_[e]	w_[e]
1	2	8
1	4	5
2	1	6
2	3	7
3	2	3
3	4	2
4	1	4
4	3	1

図 4.3 ソートしたグラフデータ

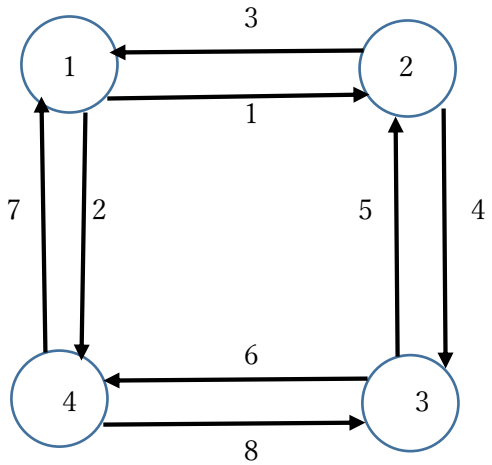

図 4.4 ソート後のグラフ

ノード v の番号	tail_[e]=v である e の番号の最大値
0	0
1	2
2	4
3	6
4	8

図 4.5 correspT の内容

ここまでで correspT の生成までが終了した．ここからはダイクストラ法の挙動を追う．任意のノード v において，status[v]=-1,distance[v]= ∞とした後，始点 s=1 に関しては，status[1]=1,distance[1]=0 で上書きする．次に correspT[0,1]すなわち，エッジ番号 1 のe_1とエッジ番号 2 のe_2の 2 つのエッジにだけについて，head_[e_1],head_[e_2]のノード x の distance[x]を更新する．この結果，distance[head_[e_1]]=8 となり，distance[head_[e_2]]=5 となる．次にノード番号 2,3,4 のv_2，v_3，v_4について distance の値が最小なノードを選び出す．仮にその値が∞なら while 文を抜ける.ここではv_4が該当する．status[v_4]=1 として，次に correspT[3,4]すなわち，tail_[e]=4 に該当する e であるe_7とe_8について，distance[head_[e_7]]と distance[head_[e_8]]を更新する．後の動作は説明を省略する．

4.3 相違点と改善点

高速化前と高速化後のアルゴリズムの違いについて触れる．改良後のアルゴリズムでは，グラフデータ配列(V,E,w_[e])を tail_[e]の値を基準にしてソートし，それを元に correspT という表を作成した．これは改良前のアルゴリズムにはなく，この処理時間分ダイクストラ法の実行開始が遅れる．

だが，correspT の作成により，ダイクストラ法の処理を改良前より速くすることができる．処理を速くしている部分は，始点と最短距離が確定したノードに隣接するノードを探索する部分である．高速化前では，探索する際に必ずエッジ全てを，最短距離確定ノードまたは始点に繋がっているかどうか調べている．それに対し，高速化後のアルゴリズムでは，correspT を使い，探索する際に必ず最短距離確定ノードまたは始点に繋がっているエッジだけを調べ，繋がっていないエッジは必ず調べない．

例えば，図の 4.4 で，ノード 3 に隣接するノードを探すとする．高速化前のアルゴリズムでは，図 4.4 のエッジ全てを調べ，隣接するノードがノード 2 とノード 4 であることを把握する．だが高速化後のアルゴリズムでは，correspT の情報を用いて，エッジ 5 とエッジ 6 だけを調べ，それで隣接するノードがノード 2 とノード 4 であることを理解する．こ

の 2 つ以外のエッジは調べない．

このようにして隣接するノードを調べることで，この探索においては，道路混雑度を算出する対象のグラフの規模が大きさに関係なくダイクストラ法をより高速に実行できる．一方で改良化前のアルゴリズムは，この探索においても，グラフ規模の影響を大きく受けるアルゴリズムである．この点は改善点と言える．

第 5 章　高速化アルゴリズム 2

本章では，ダイクストラ法実行中での，最短距離が確定したノードの探索を高速化したアルゴリズムについて説明する．

5.1 アルゴリズム紹介

第 3 章の Algorithm2 の改良版を Algorithm5 とする．なお，本章のアルゴリズムは，Algorithm1 と Algorithm5 で構成される．

Algorithm5 最短距離確定ノード探索の高速化 / Dijkstra(s)

```
 1:  finish=0
 2:  for all v ∈ V do
 3:      distance[v]=∞
 4:      flag[v]=0
 5:  end for
 6:  distance[s]=0
 7:  node[0].dist=&distance[s]
 8:  node[0].num=s
 9:  flag[s]=1
10:  for all e ∈ E do
11:      if(tail_[e]   != s){continue}
12:      x=head_[e]
13:      distance[x]=w[e]
14:      finish++
15:      node[finish].num=x
16:      node[finish].dist=&distance[x]
17:      flag[x]=1
18:  end for
19:  HeapSort
20:  while(1)
21:      u=node[1].num
22:      node[1]=node[finish]
```

```
23:     finish=finish-1
24:     DownHeapSort
25:       for all e ∈ E do
26:         if(tail_[e] != u){continue}
27:         x=head_[e]
28:         if(distance[x] > distance[u]+w[e])
29:           distance[x]=distance[u]+w[e]
30:           if(flag[x]==1){UpHeapSort(x)}
31:           if(flag[x]==0)
32:             finish++
33:             node[finish].num=x
34:             node[finish].dist=&distance[x]
35:             UpHeapSort(x)
36:             flag[x]=1
37:           end if
38:         end if
39:       end for
40:     if(finish==0){break}
41:   end while
```

5.2 アルゴリズム挙動例

本節では，本章で取り上げているアルゴリズムの挙動を説明する．ただし，高速化前のアルゴリズムについては，第 4 章でその挙動を具体的に取り上げたので説明しない．また，Heapsort,DownHeapsort,UpHeapsort の動作はこの節で具体例を挙げて説明する．

Algorithm5 の挙動を追う．説明する際は，重要な配列 node の状態だけを見ていく．まず，始点 s のノード番号と distance[s]のアドレスを node[0]に代入する．これはヒープ木には表示されない．

次に始点 s に隣接するノードの情報を node 配列に入れていき，ノードの暫定の距離が短い程配列の添え字が若くなるようにする．これが Heapsort というアルゴリズムの動作である．例えば，始点 s の隣接ノード番号が，添え字の若い順に 8,45,321 だとして，distance[8]=3,distance[45]=1,distance[321]=7 とする．そうすると，上記ソート前のヒープ木の表現が図 5.1 で，ソート後の表現が図 5.2 となる．なお，ヒープ木の円形のノードの中の数値が始点からの距離を表し，その左上の四角の中の数値が，配列 node の添え字とする．これ以降のヒープ木の表現でも同様である．

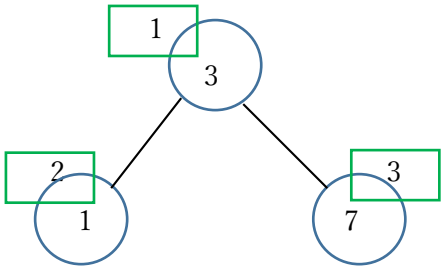

図 5.1 ソート前のヒープ木

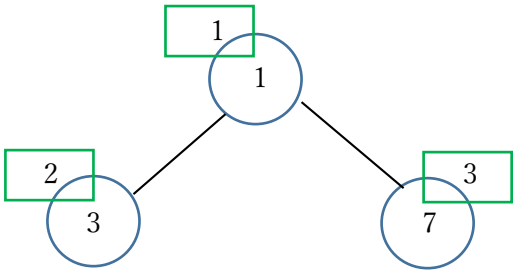

図 5.2 ソート後のヒープ木

話をアルゴリズムの挙動に戻す．上記ソートを行った後，node[1]のノードを最短距離が確定したノードとして扱う．そして，ヒープ木の中で最も深く，かつ最も右にあるノードの情報を，その根に上書きする．つまり，配列 node に格納されているノード情報の中で，最も添え字の大きい node[finish]を node[1]に代入する．例えば，図 5.2 の状態からこれを行うと，図 5.3 のようになる．とりあえずこの動作を動作 A とする．

次に，先程根の情報が上書きされたヒープ木を，暫定最短距離について親ノード≤子ノードとなるようにソートする．ただし，ヒープ木の全ノードを対象にソートするのではなく，子ノードと親ノードを交換して，元々根であったノードの深さを，整合性を保つよう深くするので，このソートに全く関わらないヒープ木のノードも存在し得る．仮に図 5.1 にこのソートを適用すると，node[1]と node[2]を交換してソートを終わらせ，node[3]は一切関わることがない．このようなソートが DownHeapsort である．例えば，根の情報が上書きされた図 5.3 ヒープ木にこのソートを適用すると，図 5.4 のようになる．

次に，動作 A で，最短距離が確定し，上書きされたノードと隣接するノードの暫定最短距離を更新する．更新の際，2 つの場合があり，場合 1 と場合 2 とする．場合 1 は，上書きされたノードと隣接するノードが，最短距離未確定かつ既にヒープ木にある場合である．この場合は，隣接するノードと同一のノードをヒープ木から探し出し，該当するノードとその親ノードを，整合性を保つために交換していく．DownHeapsort と同様に交換していくので，このソートに関与しないヒープ木のノードも存在し得る．この動作が UpHea

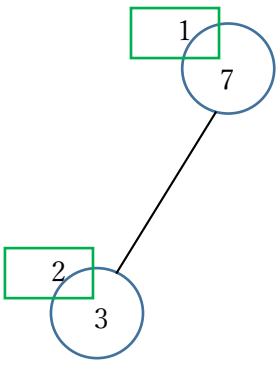

図 5.3 最短距離確定後のヒープ木

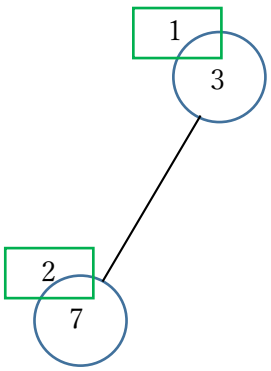

図 5.4 DownHeapsort 適用後ヒープ木

psort(x)である．x は最短距離が確定したノードに隣接するノードである．例えば，図 5.4 の状態で UpHeapsort(x)を実行し，node[2]のノードの暫定最短距離が 2 に更新されとすると，図 5.5 のようなヒープ木となる．場合 2 は，上書きされたノードと隣接するノードが，最短距離未確定かつヒープ木に入っていない，すなわち配列 node に入っていない場合である．この場合は，ノード情報を，添え字を 0 から連続させて入れる node に追加して，それから追加したノードに対して UpHeapsort(x)を行う．つまりヒープ木の整合性を保つために交換していくノードの対象にこの追加したノードが入っているのである．また ,node に追加するので，ヒープ木のノードを増やしてからソートするということである．例えば，図 5.4 の状態から，node[3]から node[5]に隣接するノードの情報を入れ，暫定最短距離が node[3]から順に 4,5,6 であるとすると，それぞれに UpHeapsort(x)を実行して図 5.6 のようになる．

あとは，ダイクストラ法が終わるまで動作 A からここまでを繰り返せばよい．

5.3 相違点と改善点

この節では，高速化前のアルゴリズムと高速化後のアルゴリズムの相違点と，高速化後のアルゴリズムに変更することによる改善点について述べる．

まずは相違点について記述する．重要かつ最大の相違点は，最短距離が確定するノードの見つけ方である．改良前では，該当するノードを見つける度に必ず全ノードを調べる．一方，改良後では，node[1]つまりヒープ木の根の情報を調べるだけで終わる．他の相違点は，改良前にはないコードが改良後のアルゴリズムにあるということである．具体的には，Node 型配列の node や Heapsort などのソートアルゴリズムなどである．

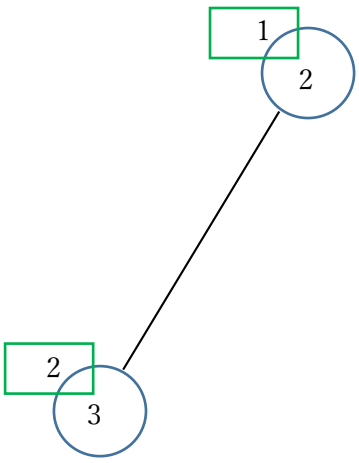

図 5.5 UpHeapsort(x)の実行（場合 1）

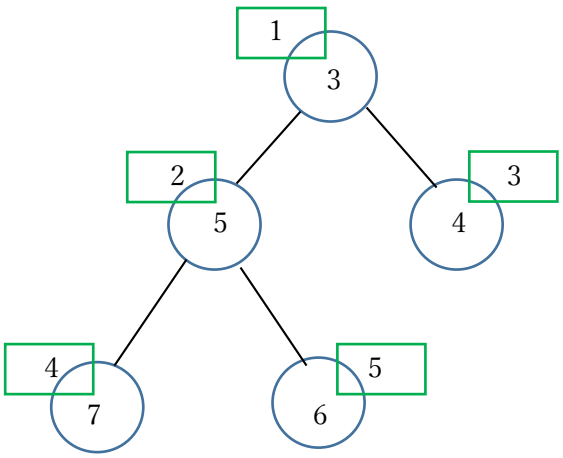

図 5.6 UpHeapsort(x)の実行（場合 2）

ここからは改善点について記述する．最短距離確定ノードをより速く見つけられることだ．配列 node や flag に値を代入したり,node の要素を交換する作業は高速化前の Algorithm2 にはなく，それに時間を費やさなければならないが，改良後のアルゴリズムでは，最短距離確定ノードを node[1]にアクセスするだけで得られるので，道路混雑度算出対象のグラフの規模が余程小さくない限り，ダイクストラ法全体では高速化後のアルゴリズムの方が速い．グラフの規模に関係なく，node[1]にアクセスするだけなので，規模が大きくなればなる程，実行時間に差が出てくる．

第 6 章　高速化アルゴリズム 3

本章では，ダイクストラ法で求めた最短経路を元に算出する，道路混雑度の計算の高速化のアルゴリズムを説明する．

6.1 アルゴリズム紹介

Algorithm1 の改良版を Algorithm6 とし，Algorithm2 の改良版を Algorithm7 とする．

Algorithm6 道路混雑度計算高速化アルゴリズム

```
 1:  READ(V,E,w_[e])
 2:  for all v ∈ V do
 3:    Execute Dijkstra(v)
 4:    for all v′ ∈ V do
 5:      if(descendant[v′]!=0){continue}
 5:      TF(descendant,v′)
 6:    end for
 7:  end for
```

Algorithm7 道路混雑度計算高速化アルゴリズム　/ Dijkstra(s)

```
 1:  for all v ∈ V do
 2:    status[v]=-1
 3:    distance[v]=∞
 4:    descendant[v]=0
 5:    father[v]=-1
 6:  end for
 7:  status[s]=1
 8:  distance[s]=0
 9:  for all e ∈ E do
10:    if(tail_[e]  != s){continue}
11:    x=head_[e]
12:    father[x]=s
13:    distance[x]=w[e]
```

```
14:  end for
15:  while(1)
16:    dist_min = ∞
17:    u=-1
18:    for all v ∈ V do
19:      if(status[v]==1){continue}
20:      if(dist_min > distance[v]){dist_min= distance[v]}
21:      u=v
22:    end for
23:    if(u==-1){break}
24:    descendant[father[u]]=1
25:    status[u]=1
26:    for all e ∈ E do
27:      if(tail_[e] != u){continue}
28:      x=head_[e]
29:      if(distance[x] > distance[u]+w[e]){distance[x]=distance[u]+w[e]}
30:      father[x]=u
31:    end for
32:  end while
```

6.2 アルゴリズム挙動例

この節で高速化前と後のアルゴリズムの挙動の詳細を見ていく．ダイクストラ法で最短経路を求め，それを元に道路混雑度を算出していくが，最短経路は求まっており，それを元に算出する場面の挙動を説明する．道路混雑度算出対象のグラフと，その最短経路の図 6.1 を以下に示す．ただし，赤く塗りつぶされているノードが始点で，赤いエッジは最短経路を構成するエッジであるとする．また，本来は，2 点間を繋ぐエッジは互いから見て

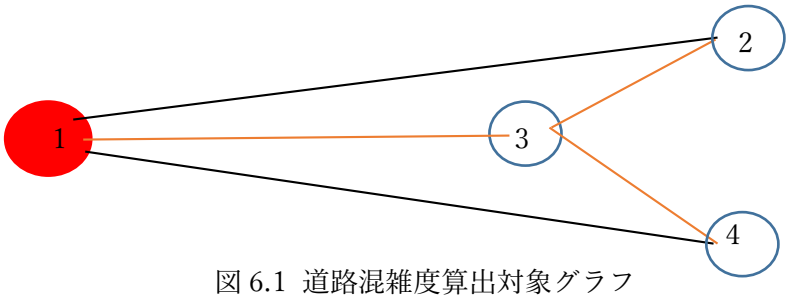

図 6.1 道路混雑度算出対象グラフ

180 度逆に向いている 2 つの矢印で表現するのが，読み込むグラフを表す厳密な表現であるが，説明しやすくするために無方向グラフで表している．

まず，高速化前の道路混雑度算出方法について見ていく．始点以外の全てのノードについて，各ノードから最短経路上をたどっていき，その際に通過したエッジの道路混雑度に 1 を足す．これを Algorithm1 が終了するまで行い，正式な道路混雑度が求まる．

この算出方法における，図 6.1 のグラフの道路混雑度計算の流れを見ていく．まずノード 2 から始点へたどる．ノード 2,3 間のエッジとノード 1,3 間のエッジを通るため，この 2 つのエッジそれぞれの道路混雑度に 1 を足す．次にノード 3 から始点へたどる．ノード 1,3 間のエッジの道路混雑度に 1 を足し，ここまででこの道路混雑度は 2 となる．最後にノード 4 から始点へたどる．ノード 3,4 間のノードとノード 1,3 間のエッジの道路混雑度に 1 を足す．ここまでで，道路混雑度に関して，ノード 1,3 間は 3 となり，ノード 2,3 間とノード 3,4 間は 1 となる．あとは Algorithm1 が終了するまでこれをすれば良い．

ここからは高速化後のアルゴリズムの挙動を説明していく．最短経路というグラフで，次数が 1 である始点以外のノードから始点へ最短経路上をたどる．たどる際は，通過したエッジの道路混雑度に，出発点からのエッジの通過数を足し，通過したノードを記憶していく．ただし，既に一度通過したノードに到達したら，それより後は通過するエッジの道路混雑度に足す値を増やさずそのままにする．これを Algorithm1 が終了するまで行い，正式な道路混雑度が求まる．

この算出方法における，図 6.1 のグラフの道路混雑度計算の流れを記述する．全ノードと赤いエッジで構成される最短経路グラフの，次数が 1 である始点以外のノードはノード 2 とノード 4 である．まずはノード 2 から始点へたどる．ノード 3 に到達した際，通過した全てのエッジ数は 1 なので，ノード 2,3 間のエッジの道路混雑度は 1 となる．そしてノード 3 を一度通過したノードとして記憶する．ノード 3 から始点に到達した際，通過した全てのエッジ数は 2 であるため，ノード 1,3 間のエッジの道路混雑度を 2 とする．

次にノード 4 から始点へたどる．ノード 3 に到達し，ノード 4,3 間のエッジの道路混雑度に 1 を足す．ただし，ノード 3 は，ノード 2 からたどる際に既に 1 度通過しているので，ノード 3 を出て行ってからは道路混雑度に足す値は 1 で固定する．そして始点に到着した際に，ノード 1,3 間のエッジの道路混雑度に 1 を足す．結果，道路混雑度において，ノード 1,3 間は 3 となり，ノード 2,3 間とノード 4,3 間は 1 となる．これを Algorithm1 が終了するまで行い，正式な道路混雑度が求まる．

6.3 相違点と改善点

重要かつ最大の違いは，道路混雑度算出方法の違いである．改良前のアルゴリズムでは，始点以外の全ノードから始点へなぞる必要があるのに対して，改良後のアルゴリズムでは，最短経路グラフの，次数が 1 である始点以外のノードから，始点へなぞれば良い．

この方法が唯一の改善点である．改良前のアルゴリズムに配列 descendant を加える必要があるが，読み込むグラフの規模が大きくなる程改良後のアルゴリズムの方が速くなる．

6.2 で，図 6.1 を対象に算出方法を説明したが，この説明では高速化後のアルゴリズムの速さを理解し辛いかもしれません．仮にノード数 1 万で，高速化前のアルゴリズムでは 1 つのノードあたり，始点へたどり着くまで平均して 30 本のエッジを通るとする．適当に始点を 1 つ定めて，その他全てのノードから始点へなぞって，道路混雑度に足す回数を求めます．9999 のノードから始点へなぞり，平均して 30 本エッジを通過するので，9999 ×30=29 万 9970 回道路混雑度に 1 を足す．

では，改良後のアルゴリズムでも足す回数を求める．先程と同じグラフを使い，また始点は先程と同じにする．400 のノードから始点へたどり，平均して 100 本エッジを通るとしする．この場合，400×100=4 万回である．

改良前と改良後では，このグラフの規模で，足す回数にこれだけの差がつく．改良後ではこの 4 万回の他に，道路混雑度に加える値を増やすかどうかの判断や，一度到達したノードとして記憶する処理などをするが，実際に測定してみても改良後の方が速い．

第 7 章　処理時間比較

本章では，道路混雑度算出対象のグラフの規模ごとに，高速化前のアルゴリズムと高速化後のアルゴリズムの計算時間を比べる．ただし，比較対象の高速化後のアルゴリズムには 3 種類ある．1 つ目が第 4 章で紹介したアルゴリズムであり，これ以降の図でアルゴリズム{4}と表す．2 つ目は第 4 章と第 5 章を組み合わせたアルゴリズムで，これ以降の図でアルゴリズム{4,5}と表す．最後の 3 つ目は第 4,5,6 章全てを組み合わせたアルゴリズムで，完成版アルゴリズムと呼ぶことにする．これ以降の図でアルゴリズム{4,5,6}と表す．また，計算時間は秒単位で表す．

7.1 実験環境

数値実験の環境は表 1 の通りである．

表 1 実験環境

CPU	Intel(R) Core(TM) i5-6200U 2.30GHz
OS	Microsoft Windows8.1 Enterprise
Memory	4.0GB
言語	C

7.2 小規模グラフの場合

ノード数 1923，エッジ数 5016 のグラフを小規模グラフとする．このグラフにおける，改良前と改良後のアルゴリズムの計算時間を図 7.1 に表示する．また，道路混雑度を可視化した図を図 7.2 に表示する．この図の縦軸の値は緯度で，横軸の値は経度である．

改良前アルゴリズム	53 秒
アルゴリズム{4}	20 秒
アルゴリズム{4,5}	2 秒
アルゴリズム{4,5,6}	1 秒

図 7.1 小規模グラフの場合の計算時間

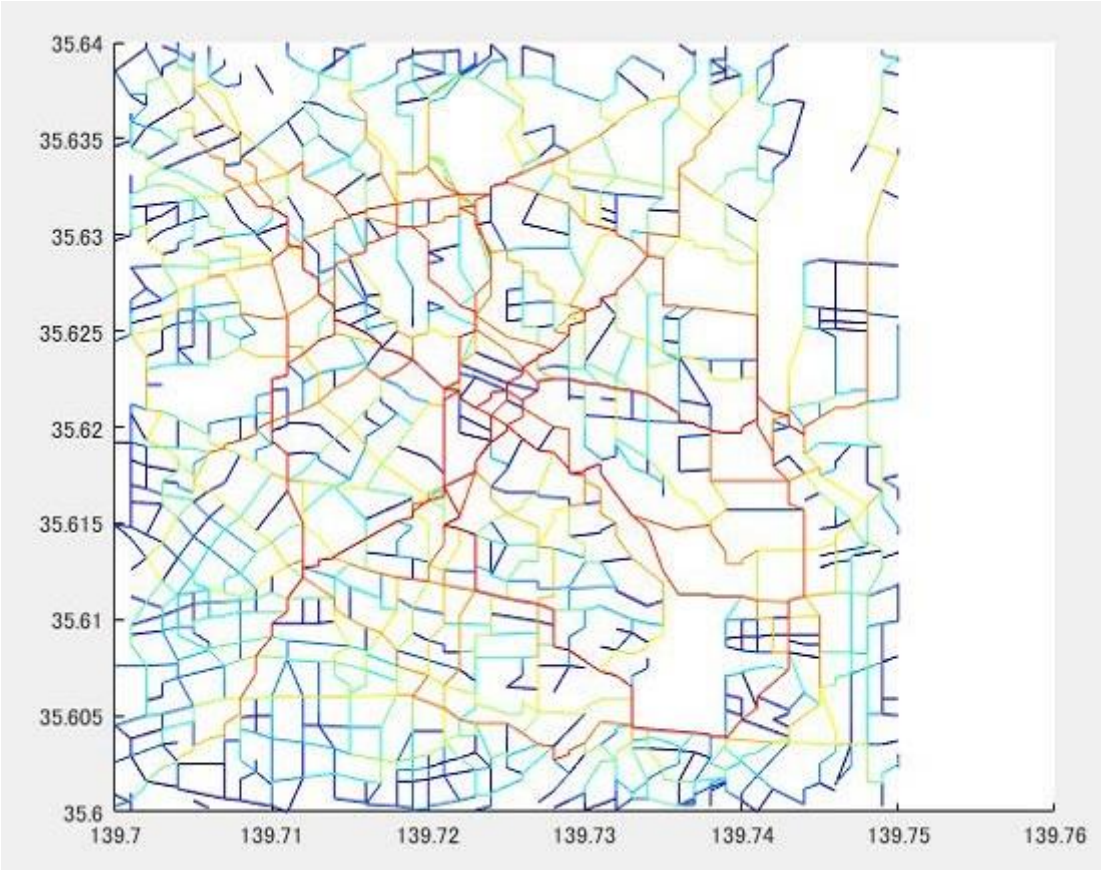

図 7.2 小規模グラフの道路混雑度の可視化

7.3 中規模グラフの場合

ノード数 3444，エッジ数 9052 のグラフを中規模グラフとする．このグラフにおける，改良前と改良後のアルゴリズムの計算時間を図 7.3 に表示する．また，道路混雑度を可視化した図を図 7.4 に表示する．この節でも，この図の縦軸の値は緯度で，横軸の値は経度である．

改良前アルゴリズム	301 秒
アルゴリズム{4}	114 秒
アルゴリズム{4,5}	6 秒
アルゴリズム{4,5,6}	4 秒

図 7.3 中規模グラフの場合の計算時間

図 7.4 中規模グラフの道路混雑度の可視化

7.4 大規模グラフの場合

ノード数 6493，エッジ数 17902 のグラフを大規模グラフとする．このグラフにおける，改良前と改良後のアルゴリズムの計算時間を図 7.5 に表示する．また，道路混雑度を可視化した図を図 7.6 に表示する．この節も同様，この図の縦軸の値は緯度で，横軸の値は経度である．

改良前アルゴリズム	2,068 秒
アルゴリズム{4}	748 秒
アルゴリズム{4,5}	26 秒
アルゴリズム{4,5,6}	15 秒

図 7.5 大規模グラフの場合の計算時間

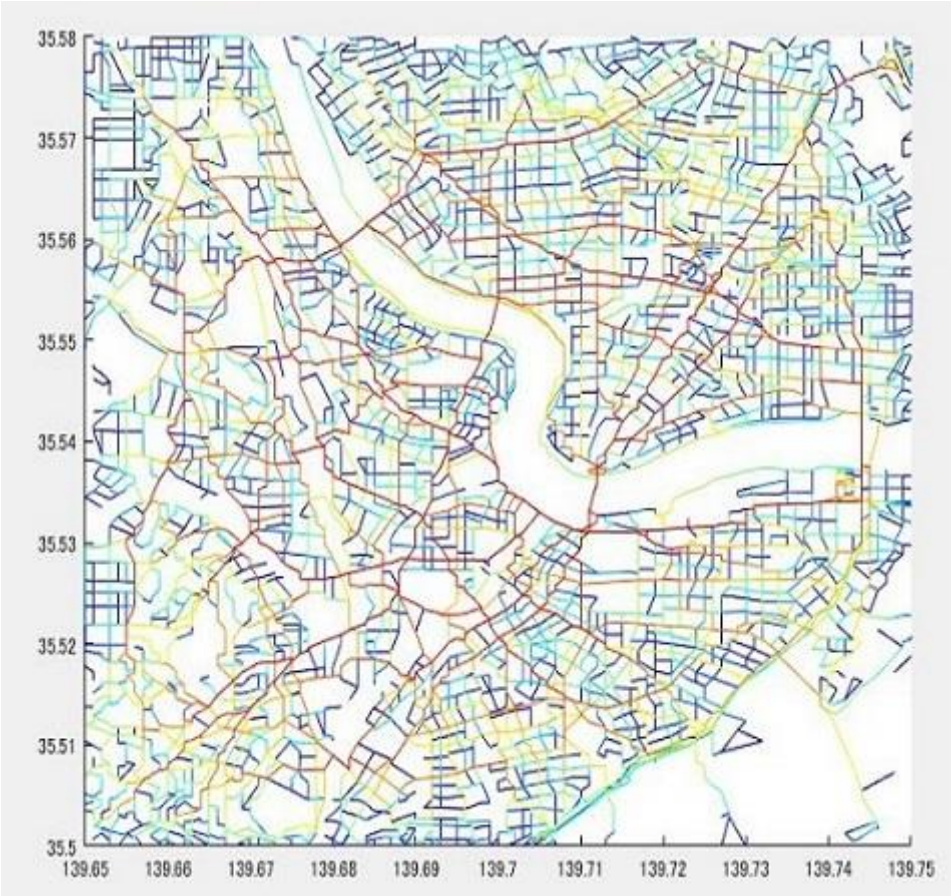

図 7.6 大規模グラフの道路混雑度の可視化

7.5 東京湾周辺の場合

ノード数 68358，エッジ数 174126 の，東京湾周辺に対して道路混雑度を算出する．算出対象の規模が大きいのでアルゴリズム{4,5,6}のみで計算時間を測定した．測定結果は 3,811 秒である．道路混雑度を可視化した図を図 7.7 に表示する．この節も同様，この図の縦軸の値は緯度で，横軸の値は経度である．

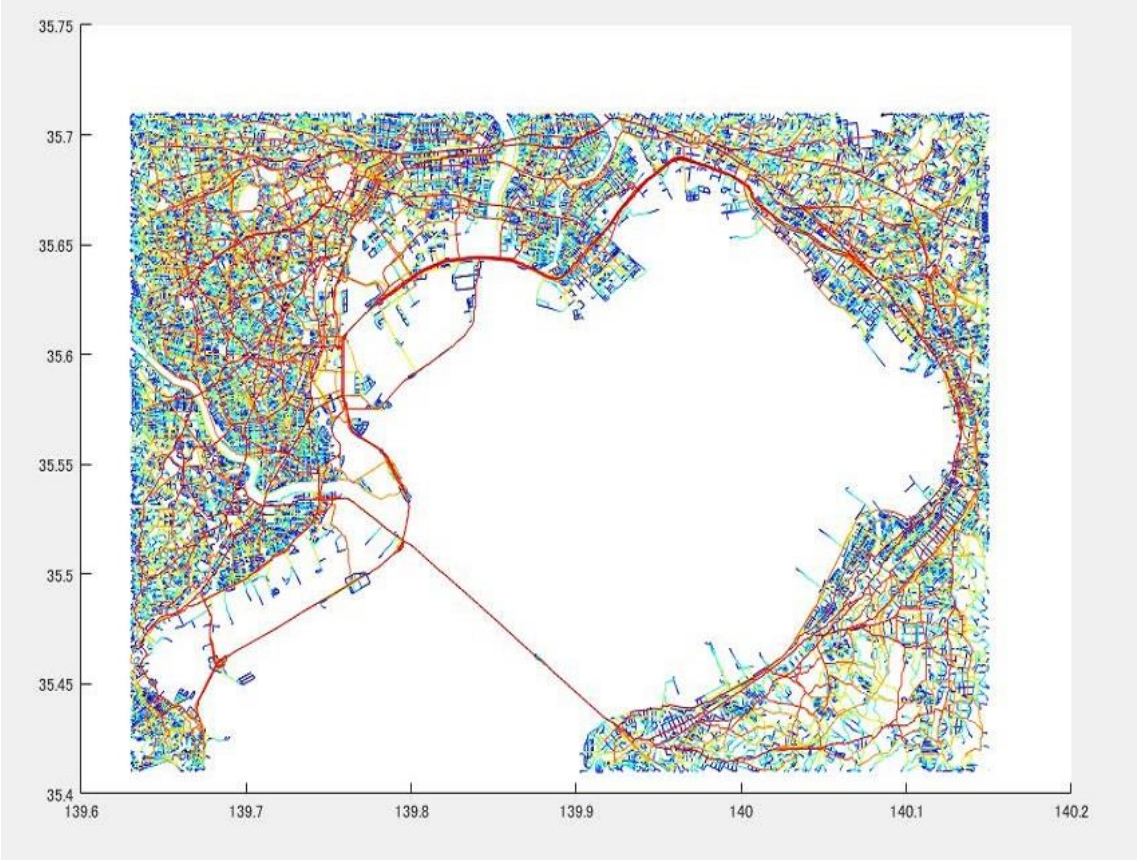

図 7.7 東京湾周辺の道路混雑度の可視化

第 8 章　考察

この章では，前章の情報を元に考察する．

提案した 3 つの改良後アルゴリズムは，本当に改良前アルゴリズムより高速かどうかを考察する．結論から記述すると，3 つ全て高速化に成功していると考えられる．なぜなら，前章の 3 つ全ての規模のグラフでの，道路混雑度の計測において，本論文で提案した高速化アルゴリズムを採用する数がより多いアルゴリズムが，採用する数がより少ないアルゴリズムより高速であることが示されているからである．

また，前章の計測結果から，グラフの規模が大きくなる程，アルゴリズム{4,5,6}すなわち完成版アルゴリズムと高速化前アルゴリズムの計算速度に差がでることが推測できる．勿論完成版アルゴリズムの方がより速いが，小規模グラフの場合は 53 倍速く，中規模グラフの場合は約 75 倍速く，大規模グラフでは約 138 倍速い．

一方で，前章の計測結果から，アルゴリズム{4}と高速化前のアルゴリズムの計算速度の差は，グラフの規模にあまり影響されない可能性があることも分かる．実際，小規模グラフではアルゴリズム{4}の方が丁度 2.65 倍速く，中規模グラフではアルゴリズム{4}の方が約 2.64 倍速く，大規模グラフではアルゴリズム{4}の方が約 2.76 倍速い．

第９章　まとめ

本研究では，ダイクストラ法自体の高速化と，ダイクストラ法で求まる最短経路を用いて計算する道路混雑度の算出時間の短縮に取り組み，合計３つのアルゴリズムを提案した．

１つ目は，ダイクストラ法実行中における，最短距離が確定したノードに隣接するノードを速く見つけるアルゴリズムを提案した．道路混雑度を算出するグラフの規模を変えて計算時間を測ったところ，このアルゴリズムと高速化前アルゴリズムの算出速度の差は，グラフの規模によらず一定である可能性があることが分かった．

２つ目は，ダイクストラ法実行中での，最短距離が確定したノードの探索を高速化するアルゴリズムを提案した．最短距離確定ノードの探索の速度は，このアルゴリズムの方が高速化前アルゴリズムより速いことは，第７章の算出時間の結果を見れば明らかである．

３つ目は，ダイクストラ法で求めた最短経路を元に算出する，道路混雑度の計算を高速化するアルゴリズムを提案した．こちらも，道路混雑算出の速度において，第７章の算出時間を見れば，高速化前アルゴリズムより速いことが分かる．

参考文献

[1] 松本俊，久保智英，井澤修平，池田大樹，高橋正也，甲田茂樹：”トラックドライバーの過労に影響する働き方と休み方の横断的検討”，労働安全衛生研究，vol.13,no.1,p1,2020

[2] 友枝明保：“「渋滞」を分析する数理モデルとしてのセルオートマンとその応用”，システム/制御/情報，vol.63,no7,p1,2019

[3] 国土交通省：道路交通センサスのデータ収集の現状と課題.
https://www.mlit.go.jp/road/ir/ir-council/ict/pdf01/03.pdf
（参照 2021 年 11 月 30 日）

[4] 国土交通省：箇所別基本表及び時間帯別交通量表に関する説明資料.
https://www.mlit.go.jp/road/census/h27/data/pdf/kasyorep.pdf
（参照 2021 年 11 月 30 日）

[5] NEXCO 東日本：高速道路の渋滞情報ってどうやって調べているの？
https://www.driveplaza.com/safetydrive/mamechishiki/029.html
（参照 2021 年 11 月 30 日）

[6] 安井雄一郎，藤沢克樹，笹島啓史，後藤和茂：“大規模最短路問題に対するダイクストラ法の高速化”，日本オペレーションズ・リサーチ和文論文誌，vol.54,pp.1-5,2011

[7] 宮崎修一：「グラフ理論入門」，森北出版 （2015）