最短路問題に対する実験的解析

修 士 学 位 論 文

最短路問題に対する実験的解析

2021 年度
（2022 年 3 月）

東京海洋大学大学院
海洋科学技術研究科
海運ロジスティクス専攻

権　鐘浩

目 次

図 目 次

第1章　はじめに

1.1　研究背景

最短経路問題は、グラフ理論やアルゴリズム設計の理論研究において古典的な問題であるだけでなく、多くの応用産業において重要な問題である。最短経路問題は、解くべき対象によって、単一始点最短経路問題、2 頂点対最短経路問題、全点対最短経路問題などに分類される。ダイクストラ法は、非負の枝長を持つグラフに対して効率的なアルゴリズムである。最短経路アルゴリズムは、旅行線路、物流計画、カーナビなど幅広い分野で応用されている。これらのシステムでは、ある点から他の点への最短経路を大量かつ高速に計算することが求められている。通常のダイクストラ法では、「次の探索点の選択」という探索処理に多くの時間がかかり、大規模なグラフを解くには適していない。この問題に対して探索候補点集合に優先度付きキューと呼ばれるデータ構造を適用することで改善することができる。decrease-key と呼ばれるものは、優先度付きキューで使用することが可能である。decrease-key を用いれば、アルゴリズムの効率はさらに向上するが、一概にそうとは言えないという。

1.2　研究目的

本研究では単一始点最短路問題に基づき、優先度付きキューを用いたダイクストラ法について、decrease-key の有無による実験的に比較し分析するものである。データの構造と実行時間を通じて解析する。その結果と実行時間をダイクストラ法による結果と実行時間と比較してどのような特徴があるかを考察する。

第2章　最短経路問題

最短路問題とは、各辺にコストが与えられているグラフ上で 2 つの頂点で結ぶ経路の中でそれに沿ったパスのコストの総和が最小となる経路を求める問題である。

2.1　ダイクストラ法

ダイクストラのアルゴリズムは、非負重みの最短経路問題を解く古典的なアルゴリズムの 1 つである。1956 年にオランダのコンピューター科学者エズゲル・ダイクストラが発見したアルゴリズムである。このアルゴリズムは多くバリエーションがある。本研究のダイクストラ法は、グラフ内の辺はすべて非負であり、ある頂点を始点としてその頂点からグラフ内のすべての頂点に対する最短経路を求めるものである。

2.2　ダイクストラ法の流れ

重み付きグラフ G=(V,E)、頂点の集合を V、辺の集合を E コスト関数 $C_i\ E \rightarrow R^+$ とする。

Step 1. 初期設定
　頂点集合 D を設定する。
　すべての頂点 v について、$dist[v] = +\infty$
　すべての頂点 v を D に追加する
　始点 s を決め、dist[s]=0

Step 2. 次の探索頂点を選択する
　集合 D は空集合ではない場合：
　　$u \leftarrow u$ が集合 D の中にあり、dist[u] は最小値である
　　u を集合 D から削除する
　集合 D が空集合の場合：終了

Step 3. 始点から頂点 v まで重みの計算と比較
　u の隣接点 v がまだ集合 D に存在する場合：
　　w = (u,v) ,$(u, v) \in E$
　　if $dist[u] + C_w < dist[v]$ then

$dist[v] = dist[u] + C_w$
Step 2 に戻る
u の隣接点 v が集合 D に存在しない場合：
Step 2 に戻る

このダイクストラ法の各反復とは全ての頂点を確認することになり、$O(|V|)$ の計算量がかかってしまう。これを頂点数だけ繰り返して計算量は $O(|V|^2)$ になる。

第3章　優先度付きキュー

前項のダイクストラ法で示した通り、頂点数が増加すると、計算量が急激に増える。優先度付きキューはダイクストラ法を高速化するための 1 つの方法である。

本研究でダイクストラ法の実装に関して使われる優先度付きキュー [1] はバイナリヒープに基づいて実装されている。高速な検索と削除など操作により、ダイクストラ法の効率を向上させることができる。

3.1　バイナリヒープ

ヒープは、データ構造の一種で、木構造のうち、要素の挿入と最大値や最小値の要素を削除する計算量が要素数を N としたとき共に O(logN) になる。このヒープを二分木で表現したものはバイナリヒープと呼んでいる。

3.1.1　二分木

二分木はデータを階層的に表現するデータ構造の 1 つである。図 3.1 のように表現される。

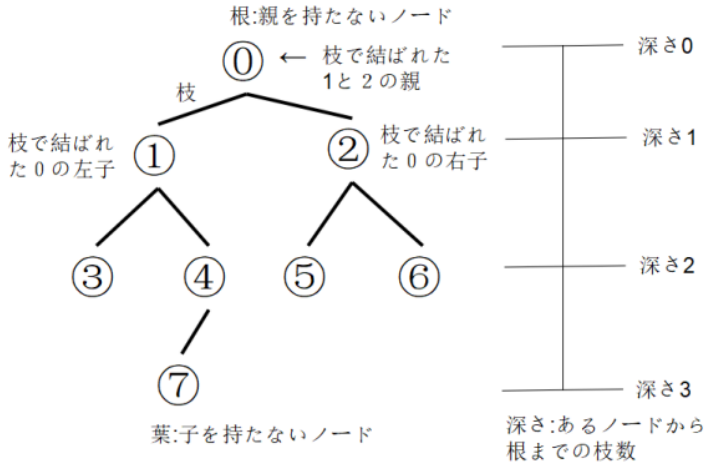

図 3.1: 二分木の例

3.1.2　バイナリヒープ化

親と子が持つ値を比較し、すべての親子関係において親が持つ値の優先度が高いという条件満たすことで二分木を使って作られるヒープはバイナリヒープである。問題によって異なり、根に最大値を与えられた場合高い値の優先度が高いである。ヒープの制約は親の値が子の値より大きくになる。逆に、根に最小値を与えられた場合小さい値の優先度が高いである。ヒープの制約はヒープの制約は親の値が子の値より小さくになる。

図 3:2 は根が最小値となるヒープであり、各キーの値は、その子の値より小さくなければならないという関係である。

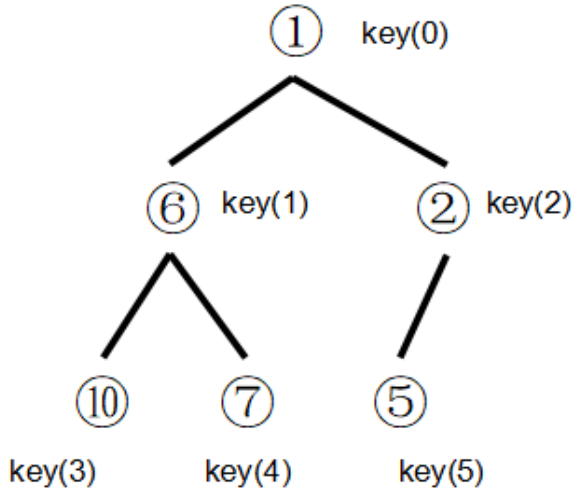

図 3.2: バイナリヒープの例

3.1.3　バイナリヒープの追加と削除

親子関係の条件を満たす新しいキーをバイナリヒープに追加するために、sift-up という操作がある。

１．新しい key をヒープ配列の末尾に配置する。
２．追加された key とその親の key を比較し、親子関係の条件を満たせば終了。
３．親子関係の条件を満たさない場合、追加された key とその親の key を入れ替えて 2 に戻す。

図 3:3 は、根が最小値を持つヒープに新しい key[3] を追加する sift-up 操作の例である。

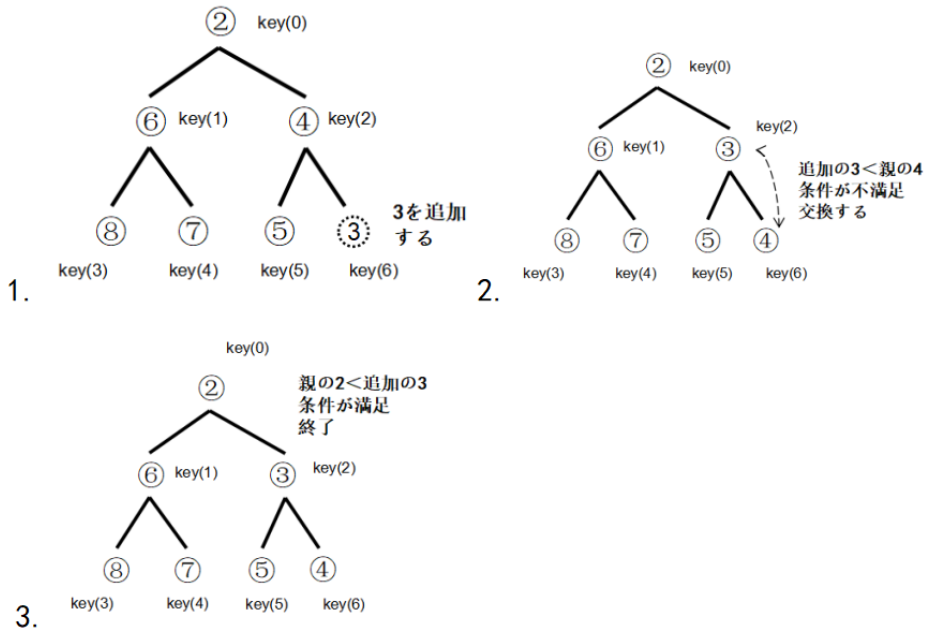

図 3.3: sift-up 操作の例

バイナリヒープからある key を削除した後、親子関係を満たすことを条件に、sift-down と呼ばれる操作が行われる。
１．削除された key の位置は、最後の key に置き換えられる。
２．比較の順序は設定に依存し、ここでは交換された key はまずその左子の key と比較され、親子関係の条件を満たした場合にはその右子の key と比較される。すべての条件を満たした場合、処理は終了である。
３．key とその子の key を比較した結果、親子関係が成立しない場合は、それら 2 つのキーが入れ替えて 2 に戻す。

図 3:4 は、根が最小値を持つヒープから根を削除した後の sift-down 操作の例である。

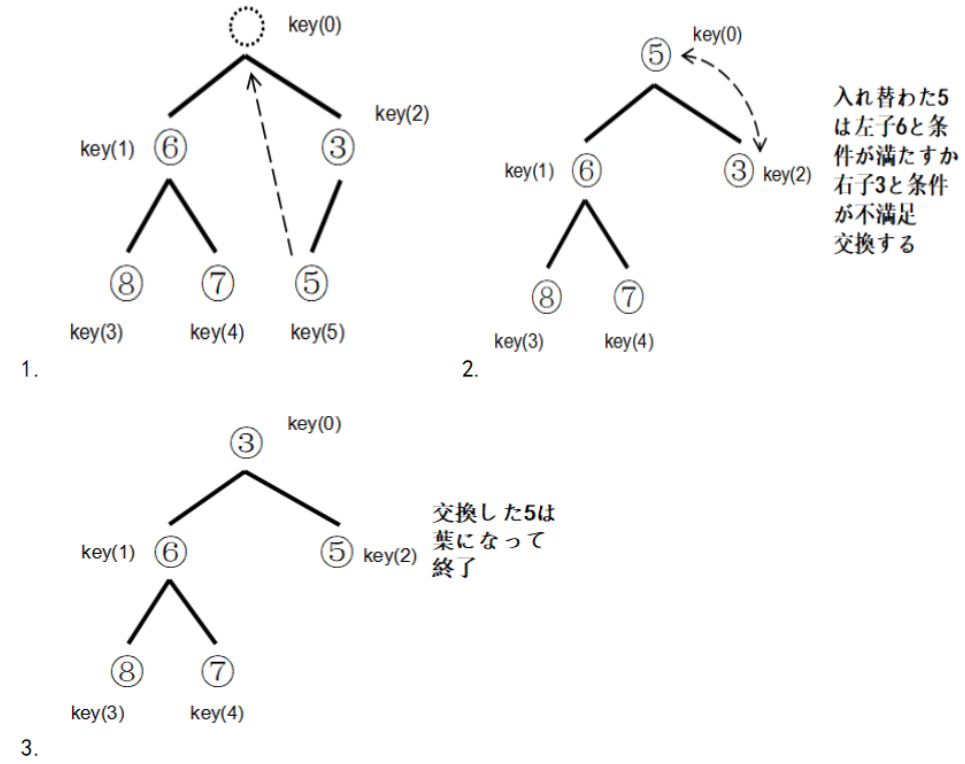

図 3.4: sift-down 操作の例

3.2 ダイクストラ法の実装

優先度付きキューを用いたダイクストラ法では、まだアクセス可能な頂点と値（出発点までの距離）は優先度付きキューに格納される。バイナリヒープの特性を利用して、最小の値に対して最高の優先度を割り当てるのである。これにより、格納された頂点と値の中から、最も小さい値を持つ頂点と値を素早く見つけることができ、効果的に時間を短縮することができる。

3.2.1 優先度付きキューを用いたダイクストラ法

優先度付きキューに頂点と値を追加するとは、優先度付きキューのヒープに頂点と値（始点からの距離）を追加し、ヒープを実装する配列の最後の位置に配置することである。その後、sift-up 操作を行うことで、正しい位置に更新される。

Step 1. 初期設定

G(V,E)　V はグラフ中のすべての頂点の集合である。
　　E はグラフ中のすべて枝の集合である。

Q = priority queue()

dist[i]=+∞ $i \in V$ (各頂点 i について、始点から頂点 i までの距離を無限大とする)

dist[s]=0　(頂点 s を始点とし、s から始点までの距離は 0 である)

$(s, 0) \rightarrow Q$　　(Q に (頂点 s, 値 0) を追加する)

Step 2. 探索頂点を選択して次の探索頂点を探す

(v,d)=Q.pop()　　(Q から削除した最小値を d とし、削除した頂点を v とする)

dist[v]=d　(頂点 v から始点までの最短距離は d である)

W = (v,u)

集合 D は空集合になる場合：終了

Step 3. 次の探索頂点を格納する

v から隣接点 u までの距離：$C_{v,u}$

if $dist[v] + C_{v,u} \leq dist[u]$ ：

　then $(u, dist[v] + C_{v,u}) \rightarrow Q$

　$dist[u] = dist[v] + C_{v,u}$

　(Q に (頂点 u, 値 $dist[v] + C_{v,u}$) を追加して、$dist[u]$ の値を $dist[v] + C_{v,u}$ する)

Step 4. 繰り返し

Q が空集合の場合、アルゴリズムは終了する。

それでない場合、Step 2 に戻る。

3.3 networkX のダイクストラ法

Python では、標準ライブラリに heapq[3] として優先度付きキューが提供されており、Python のネットワーク計算パッケージ networkX は heapq[3] を使ってダイクストラ法を実装している。

networkX のダイクストラ法のデータ構造は、本研究で使用したダイクストラ法のデータ構造とは多少異なる。前者は、始点から他のすべての頂点までの最短距離だけでなく、最短パスも格納している。後者は、最短距離のみを格納している。

第4章　decrease-key

4.1　優先度付きキューの decrease-key

decrease-key は、優先度付きキューでダイクストラ法を行う際に、追加したい頂点が既にヒープにあり、ヒープにある頂点が大きな値を持つ場合に使用できる操作である。ヒープ内の頂点の位置を探し、その頂点をもつ値を追加したい値に変更し sift-up2 操作を行い、頂点を正しい位置に配置する。

decrease-key を使用するため、3.2.1 節の Step 3 を修正すると：

Step 3. decrease-key で距離を更新する
　　if $dist[v] + C_{v,u} \leq dist[u]$ ：
　　　if u in Q:
　　　　then decrease-key(u,$dist[v] + C_{v,u}$)
　　　　Otherwise $(u, dist[v] + C_{v,u}) \rightarrow Q$
　　　dist[u]= dist[v] +W
　　(Q に頂点 u がある場合、decrease-key 操作を行いである。
　　Qに (頂点 u, 値 $dist[v] + C_{v,u}$) を追加して、dist[u] の値を $dist[v] + C_{v,u}$ する)

優先度付きキューに heap=[(1,2),(2,3)(3,4)(4,5)] がある場合。要素のは（頂点、値）である。このヒープに要素 (2,1) を追加する。同じ頂点 2 を持っているので、decrease-key 操作は図 4.1 に示す。

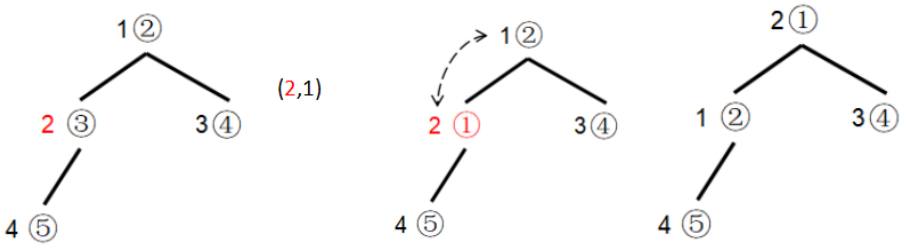

図 4.1: decrease-key 操作の例

4.2 Heap dict

Heap dict[4] は、Daniel Stutzbach が作ったコードである。基本的にはバイナリヒープとして実装された優先度付きキューである。Heap dict[4] と本研究の優先度付きキュー [１] との違いは、Heap dict[4] は要素を追加した後に decrease-key 操作を自動的に行う点である。

4.2.1 Heap dict の decrease-key を解読

Heap dict[4] を使用する場合、追加したい頂点がすでに Heap dict[4] のヒープにある場合、その頂点はまずヒープから削除する。そして、追加される頂点と値はヒープの末尾に配置する。親子関係を満たすために、sift-up 操作を行い、頂点を正しい位置に更新する。自動実行されるため、追加したい頂点の値をヒープ内の同じ頂点が持つの値より小さく設定してから追加する必要がある。

heap dict[4] の heap=[(3,2),(2,4),(4,5),(1,6),(5,7),(6,10)]。要素のは（頂点、値）である。そのヒープに要素（2,3）を追加する処理は、図 4.2、図 4.3 に示す。

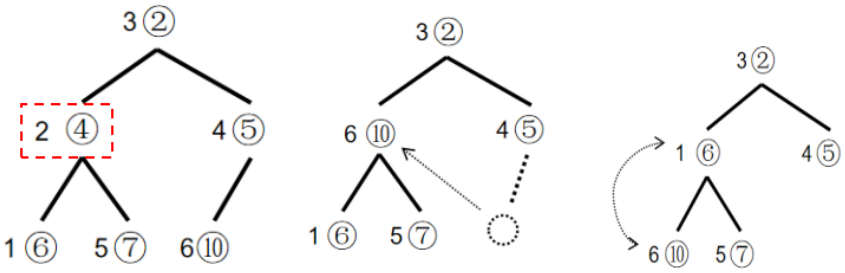

図 4.2: Heap dict 例 (1)

追加したい頂点が既にヒープに存在することを確認し、先にヒープから削除して sift-down 操作を行いである。

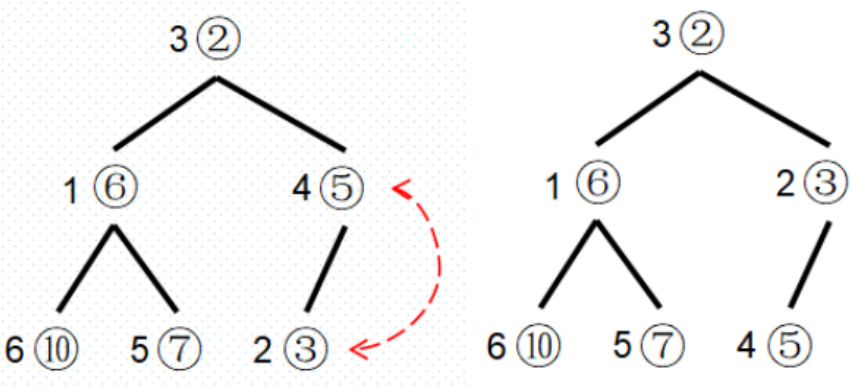

図 4.3: Heap dict 例 (2)

追加された要素をヒープの最後の位置に配置して、Sift-up の操作が実行される。

4.3 ダイクストラ法への影響

decrease-key は、ヒープに余分な情報を追加しないため、また、既存のオブジェクトの優先度を変更できるようにすることで、優先度付きキューを用いたダイクストラ法の効率を改善するために使用することができる。

第5章　実験

本研究の実験は、一対全最短距離問題の decrease-key の有無に対する比較実験である。

5.1　実験環境

プログラミング言語：　python
os 種類：Windows　64 ビット
プロセッサ：　Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz
RAM : 8.00GB

5.2　decrease-key について比較実験

本比較実験では、4 つのバージョンの優先度付きキューを比較対象としている。 第 1 列は decrease-key を行うもの、第 2 列は decrease-key を行わないもの、第 3 列は自動的に decrease-key を行う Heap dict[4] を使用するもの、第 4 列は decrease-key なしの heapq[3] である。グラフのある頂点を始点としてダイクストラ法を実行する。同じ始点で 5 つの実験の結果を平均し、ダイクストラ法の各バージョンについてかかった時間を測定し、異なる実験で比較を行うものである。

5.2.1　実験結果

実験データの最初の数桁 (表 5.1) は networkx によってランダムに生成されたグラフである。頂点数 264,346 からアメリカの実際の道路ネットワークを使用したものである。図 5.1、図 5.2 と図 5.3 は頂点数 264,346 の道路ネットワークグラフを利用して実験したものであり、優先度付きキュー [1] を利用した 2 つバージョンの各コードの時間割を表示されるものである。実行時間 (表 5.1) の単位は秒で表示している。

表 5.1: python におけるダイクストラ法の性能 (実行時間)

頂点数	decrease-key	no-decrease-key	heap dict	heapq
10	0.0003	0.00019	0.00021	0
100	0.0019	0.0017	0.0023	0.00178
1,000	0.0255	0.0318	0.0261	0.01457
10,000	0.29	0.35	0.27	0.1673
100,000	4.13	5.18	3.88	2.0568
264,346	5.47	6.14	4.5	3.0535
321,270	6.44	6.84	5.22	3.3374
435,666	8.601	9.058	6.809	4.482
1,070,376	23.69	24.95	19.41	11.185
1,890,815	40.4	43.45	32.65	22.345

decrease-key 有無の各バージョンの計算時間プロファイリングは図 5.1、図 5.2、図 5.3 に示す。

```
% Time  Line Contents
=====================
        def dec_heap(n,G,i):
   0.0      source = i
   0.0      node =n
   0.0      G_succ = G.succ if G.is_directed() else G.adj
   0.0      distlist=[sys.maxsize] * (node)
   0.0      distlist[source]=0
   0.0      dist={}
   0.0      Q = PriorityQueue()
   0.0      Q.insert(source,0)
   0.7      while Q.heap:
  61.2          (v, d) = Q.extract_min()
   0.9          dist[v] = d
  12.7          for u, e in G_succ[v].items():
  13.7              vu_dist = dist[v] + G[v][u]["weight"]
   2.3              if vu_dist < distlist[u]:
   1.0                  if distlist[u]==sys.maxsize:
   5.9                      Q.insert(u, vu_dist)
                        else:
   0.8                      Q.decrease_key(u,vu_dist)
   0.8                  distlist[u]=vu_dist
```

図 5.1: decrease-key(計算時間プロファイリング)

```
% Time  Line Contents
=====================
        def nodec_heap(n,G,i):
   0.0      source = 0
   0.0      node =n
   0.0      G_succ = G.succ if G.is_directed() else G.adj
   0.0      distlist=[sys.maxsize] * (node)
   0.0      distlist[source]=0
   0.0      dist={}
   0.0      Q = PriorityQueue()
   0.0      Q.insert(source,0)
   0.7      while Q.heap:
  64.5          (v, d) = Q.extract_min()
   0.9          if v in dist:
   0.1              continue
   0.7          dist[v] = d
  11.6          for u, e in G_succ[v].items():
  12.5              vu_dist = dist[v] + G[v][u]["weight"]
   2.1              if vu_dist < distlist[u]:
   6.1                  Q.insert(u, vu_dist)
   0.8                  distlist[u]=vu_dist
```

図 5.2: no-decrease-key(計算時間プロファイリング)

```
 % Time  Line Contents
=====================
        def heap_dict1(n,G,i):
   0.0      source=i
   0.0      node=n
   0.0      G_succ = G.succ if G.is_directed() else G.adj
   0.0      distlist=[sys.maxsize] * (node)
   0.0      distlist[source]=0
   0.0      dist={}
   0.0      hp=heapdict()
   0.0      hp[source] = 0
   1.6      while hp:
  49.9          (v, d) = hp.popitem()
   1.1          dist[v] = d
  14.0          for u, e in G_succ[v].items():
  14.4              vu_dist = dist[v] + G[v][u]["weight"]
   2.4              if vu_dist < distlist[u]:
  15.7                  hp[u]=vu_dist
   0.9                  distlist[u]=vu_dist
```

図 5.3: Heap dict(計算時間プロファイリング)

5.2.2 結果分析

heap dict[4] と heapq[3] の優先度付きキューのデータ構造は、本研究で使用するもの [1] とは異なるため、主に最初の 2 つのバージョンを比較する。実行時間 (表 5.1) から分析すると、グラフの頂点数が少ない (100 以下) 場合、decrease-key がないバージョンの方が、decrease-key があるバージョンの結果よりも良好であることがわかる。グラフの頂点数が多くなると、decrease-key の使用による効果が顕著になる。

全体的にみると、同じダイクストラ法のデータ構造を使用した場合、heapq[3] のバージョンは非常に良い結果を得ることになる。heapq[3] のデータ構造が他の優先度付きキューのデータ構造より優れていることが証明されている。

heapq[3] のバージョンを抜き取り、頂点数が非常に少ない (100 以下) 場合、heap dict[4] のバージョンも decrease-key なしのバージョンほど良い結果は得られなかったである。decrease-key を使用して 2 つのバージョンを比較すると、heap dict[4] バージョンの方が良い結果を得ることがである。

図 5.1、図 5.2、図 5.3 から解析すると、どちらのバージョンも次の探索頂点の選択 ((Q.extract-min()/hp.popitem()) に最も時間がかかるである。decrease-key を使用したバージョンで、次の探索頂点の選択にかかる時間を短縮できるだけでなく、優先度付きキュー [1] の探索頂点を格納すること (Q.insert()) にかかる時間も短縮できる。

5.3 メモリの変化について比較実験 1

本比較実験では、優先度付きキュー [1] の 2 つバージョンと heap dict[4] を用いたダイクストラ法のメモリの変化を測定し、異なる実験で比較を行うものである。

メモリの増量や変化を測定するため、3 つの大規模グラフのデータを使って実験を行いである。

5.3.1 実験結果

システムの影響を除去するため、数回のテストをした後、安定したデータを取って表示する。メモリの増量 (表 5.2) の単位は MiB で表示している。

表 5.2: python にダイクストラ法の性能 (メモリ増量)

頂点数	dec-heap	nodec-heap	heapdict
264,346	12.04	12.09	12.06
321,270	12.46	12.46	12.46
435,666	23.33	23.33	23.33

メモリの変化を確認するために、各バージョンの 264,346 頂点数のデータを用いた実験結果を可視化して図 5.4、図 5.5、図 5.6 に示す。

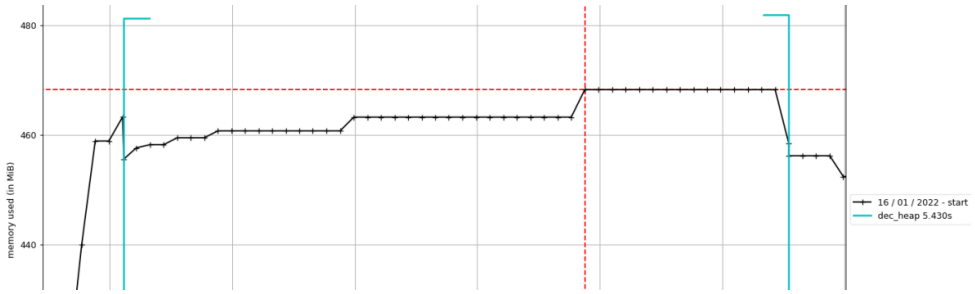

図 5.4: decrease-key のメモリ変化

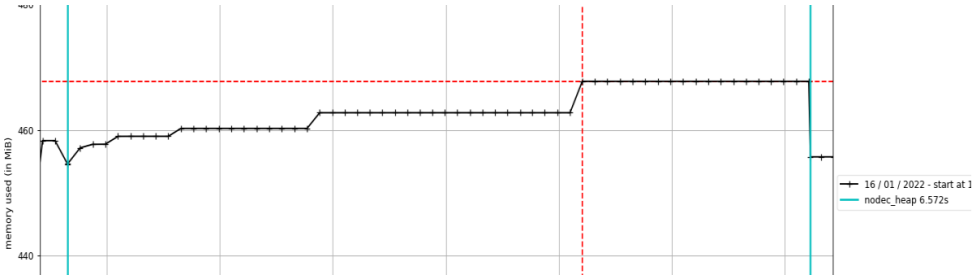

図 5.5: no-decrease-key のメモリ変化

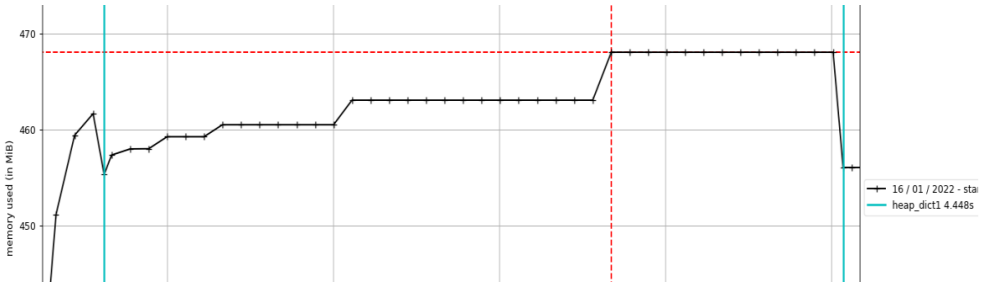

図 5.6: Heap dict のメモリ変化

5.3.2 結果分析

図 5.4、図 5.5、図 5.6 から分かると、各バージョンのメモ量はほぼ同じように変化する。decrease-key を使用しても全体のメモリ量は減らなかったが、収集したデータから分かると decrease-key ありのバージョンは decrease-key なしのバージョンにより、メモリのピークが持続する時間が短い。

5.4　メモリの変化について比較実験 2

本実験では、優先度付きキュー [1] の 3 つの部分、探索頂点の選択 (extract-min)、探索頂点の格納 (insert) と decrease-key に対し、メモリの変化を測定して実験的に比較する。

5.2.1 節の結果から分かると、頂点数が 100 以下時、decrease-key なしのバージョンは良い結果を出ている。データの処理を容易にするために、2 つのバージョンのダイクストラ法では、それぞれ 80 頂点と 500 頂点を持つ networkx でランダムに生成されたグラフを使用する。

5.4.1　頂点数 500 の実験結果

2 つバージョンの test1 の実験は同じグラフあり、他のて test の実験のグラフは同じ構造が、距離は 1 から 100 の間でランダムに生成したものである。

表 5.3: decrease-key ありのバージョン (頂点数 500)

	最初のメモリ	最終のメモリ	回数	メモリの変化量
test1				
insert	115.1562	115.5586	500	0.4024
extract_min	115.1289	115.5586	500	0.4297
decrease_key	115.1562	115.5586	154	0.4024
test2				
insert	114.9648	115.3828	500	0.418
extract_min	114.9648	115.3906	500	0.4258
decrease_key	114.9648	115.3828	175	0.418
test3				
insert	115.2227	115.6797	500	0.456
extract_min	115.2227	115.6914	500	0.4686
decrease_key	115.2852	115.6875	162	0.4023
test4				
insert	115.332	115.7461	500	0.4141
extract_min	115.332	115.75	500	0.418
decrease_key	115.332	115.7461	142	0.4141
test5				
insert	115.0742	115.4961	500	0.4219
extract_min	115.0742	115.5039	500	0.4297
decrease_key	115.0742	115.4961	156	0.4219

表 5.4: decrease-key なしのバージョン (頂点数 500)

	最初のメモリ	最終のメモリ	回数	メモリの変化量
test1				
insert	115.1523	115.6562	652	0.4609
extract_min	115.1562	115.6602	652	0.4648
test6				
insert	115.0508	115.543	646	0.4922
extract_min	115.0508	115.5508	646	0.5
test7				
insert	114.9375	115.4062	668	0.4687
extract_min	114.9375	115.4844	668	0.5469
test8				
insert	114.8984	115.3906	662	0.4922
extract_min	114.8984	115.4648	662	0.5664
test9				
insert	115.1562	115.5859	654	0.4297
extract_min	115.1562	115.6328	654	0.4766

実験対象のメモリ変化を確認するために、decrease-key 有無の 2 つババージョンの test1 の実験結果を可視化して図 5.7、図 5.8 に示す。

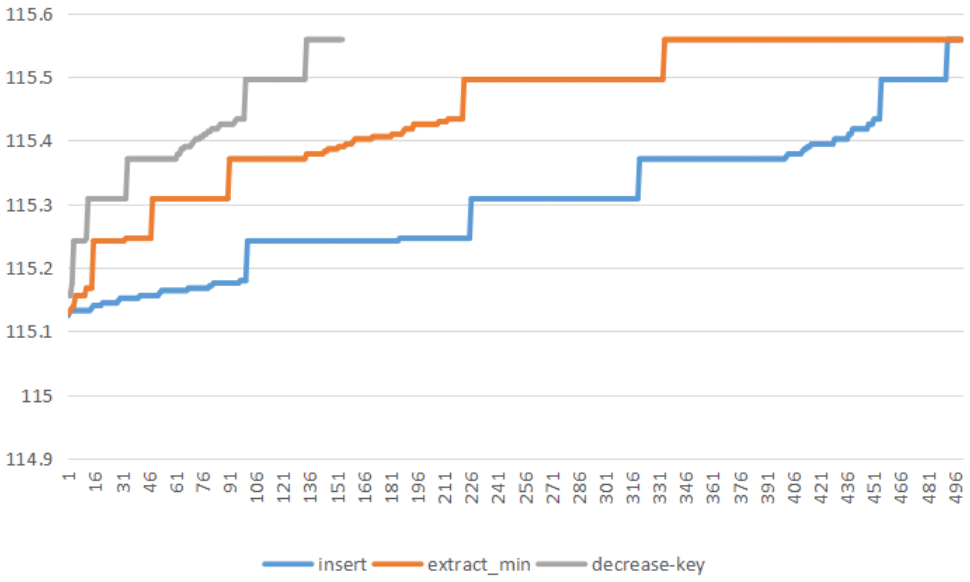

図 5.7: decrease-key あり　test1 のメモリ変化

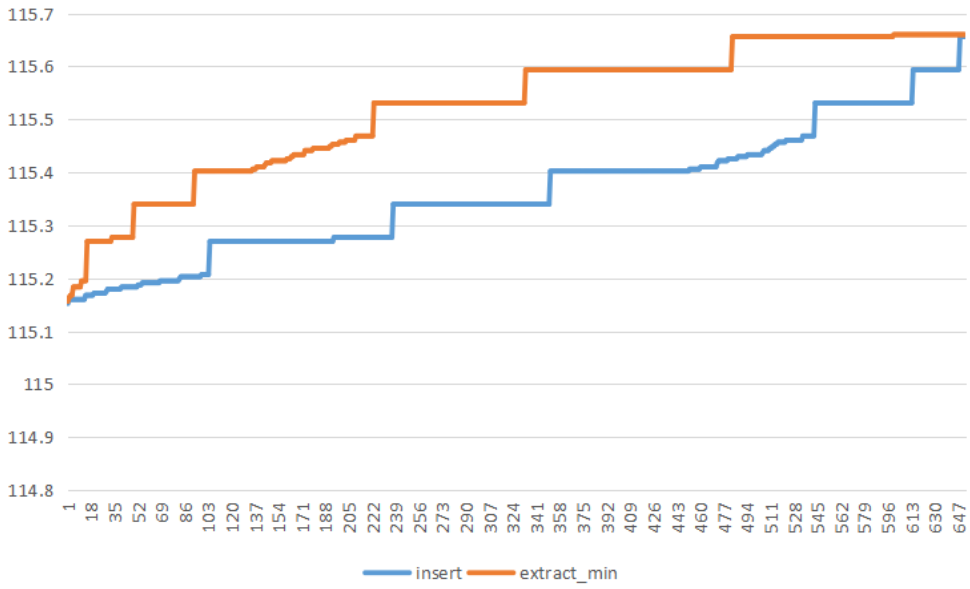

図 5.8: decrease-key なし　test1 のメモリ変化

5.4.2　頂点数 500 の結果分析

2 つのバージョンの結果を比較すると、decrease-key ありのバージョンは探索頂点の選択 (extract-min) の回数と探索頂点の格納 (insert) の回数を減らすことで計算量を減らし、結果として decrease-key なしのバージョンよりもメモリの変動量を少なくすることができる。メモリの変化については、メモリの変化量が小さいほどシステムへの影響が小さくなり、その結果アルゴリズムの実行時間が影響されると考える。

探索頂点の選択 (extract-min) と探索頂点の格納 (insert) の 2 つバージョンのメモリ変化の比較を図 5.9、図 5.10 に示す。

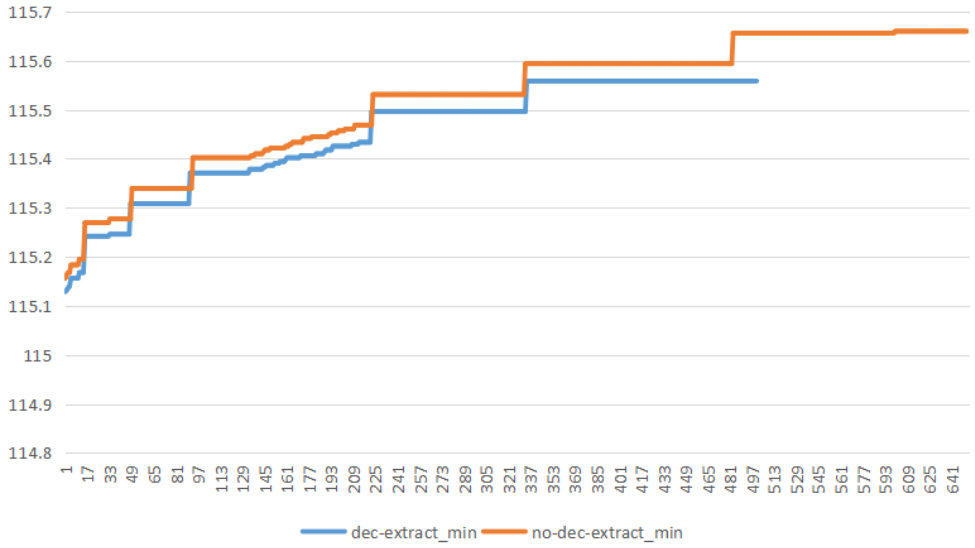

図 5.9: 探索頂点の選択 (extract-min) のメモリ変化の比較

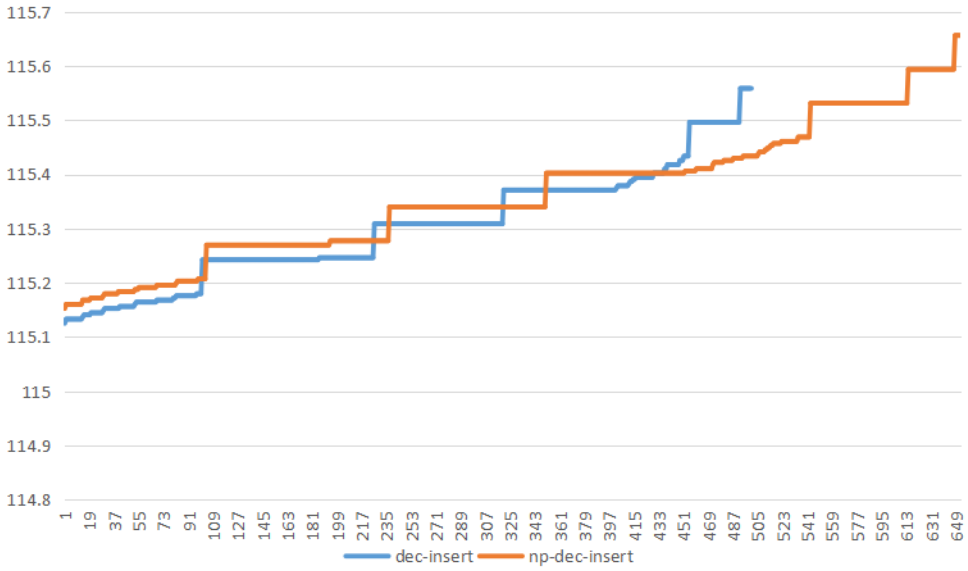

図 5.10: 探索頂点の格納 (insert) のメモリ変化の比較

5.4.3 頂点数 80 の実験結果

2 つバージョンの test5 以外の実験は同じ構造で距離は 5 から 60 の間でランダムに生成したグラフである。test5 のグラフは、枝が追加され、距離のランダムレ範囲が増加し、他のグラフより少し複雑になっている。

表 5.5: decrease-key ありのバージョン (頂点数 80)

	最初メモリ	最終メモリ	回数	メモリ変化量
test1				
insert	114.6094	114.6719	80	0.0625
extract_min	114.6094	114.6719	80	0.0625
decrease_key	114.6094	114.6719	21	0.0625
test2				
insert	114.3945	114.4414	80	0.0469
extract_min	114.3945	114.4492	80	0.0547
decrease_key	114.3945	114.4414	21	0.0469
test3				
insert	114.6875	114.75	80	0.0625
extract_min	114.6875	114.75	80	0.0625
decrease_key	114.5	114.5391	21	0.0391
test4				
insert	114.6289	114.6914	80	0.0625
extract_min	114.6289	114.6914	80	0.0625
decrease_key	114.6289	114.6914	19	0.0625
test5				
insert	114.9219	114.957	80	0.0351
extract_min	114.9219	114.9766	80	0.0547
decrease_key	114.9219	114.9648	54	0.0429

表 5.6: decrease-key なしのバージョン (頂点数 80)

	最初メモリ	最終メモリ	回数	メモリ変化量
test1				
insert	114.4688	114.5312	101	0.0624
extract_min	114.4688	114.5312	101	0.0624
test2				
insert	114.8438	114.9062	101	0.0624
extract_min	114.8438	114.9062	101	0.0624
test3				
insert	114.4062	114.4688	101	0.0626
extract_min	114.4062	114.4688	101	0.0626
test4				
insert	114.4609	114.5234	99	0.0625
extract_min	114.4609	114.5234	99	0.0625
test5				
insert	114.457	114.5078	134	0.0508
extract_min	114.457	114.582	134	0.125

実験対象のメモリ変化を確認するために、decrease-key 有無の 2 つババージョンの test1 と test5 の実験結果を可視化して図 5.9、図 5.10 と図 5.11、図 5.12 に示す。

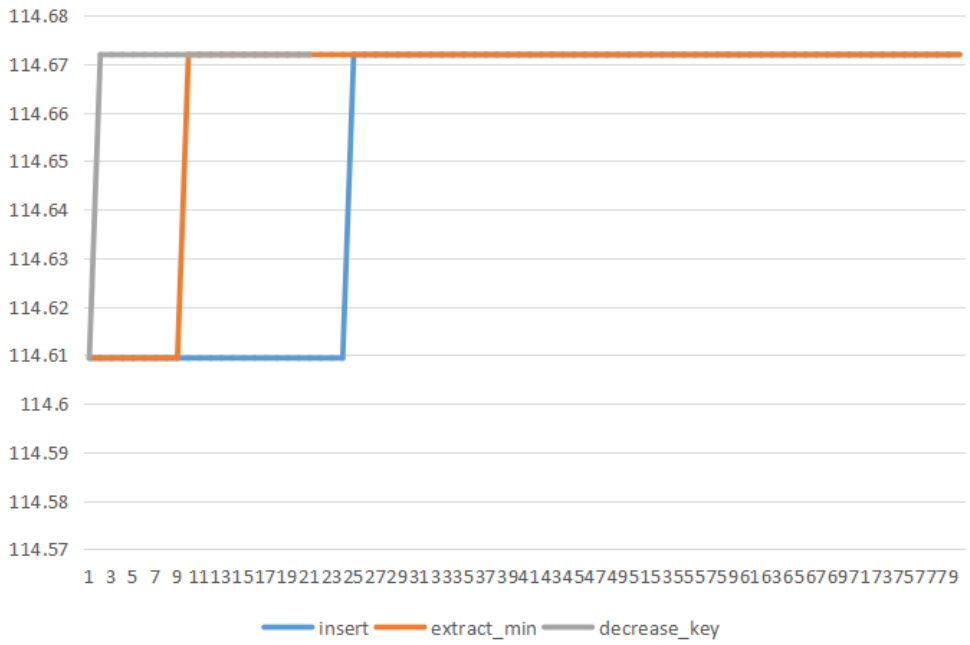

図 5.11: decrease-key あり　test1 のメモリ変化

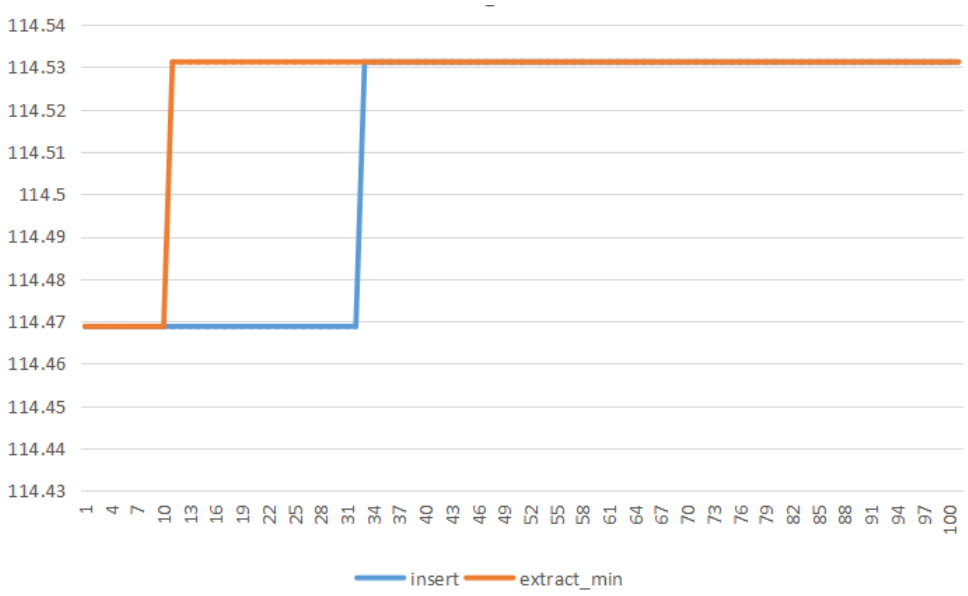

図 5.12: decrease-key なし　test1 のメモリ変化

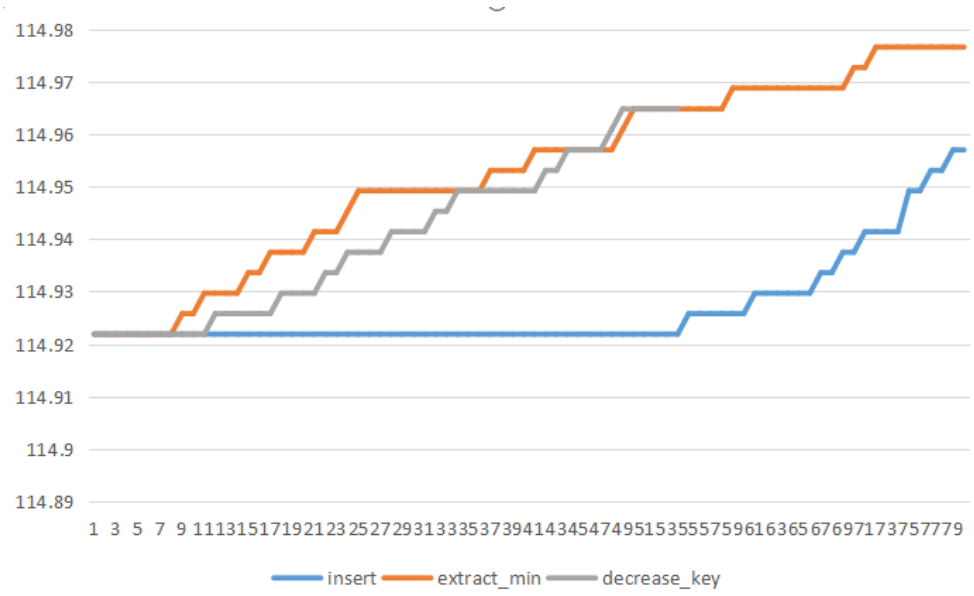

図 5.13: decrease-key あり　test5 のメモリ変化

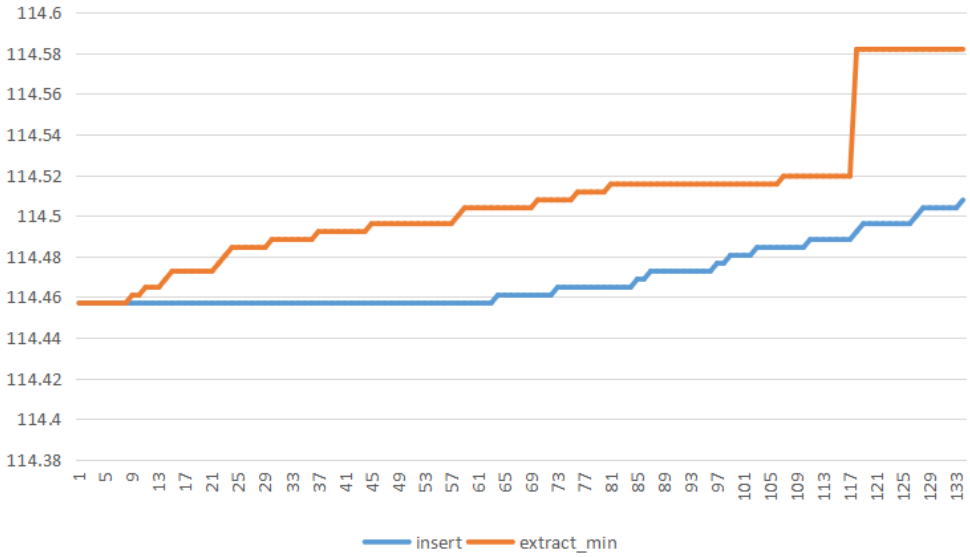

図 5.14: decrease-key なし　test5 のメモリ変化

5.4.4　頂点数 80 の結果分析

2 つのバージョンの結果を比較すると、頂点数が少ない場合、decrease-key を使っても使わなくても、探索頂点の選択と格納の探索のメモリの変化に対するあんまり影響がないである。グラフの複雑さが増すと、メモリ変化に対する decrease-key の効果が明らかになり始める。

5.5 networkX のダイクストラ法について比較実験

本比較実験では、4 つのバージョンで networkX のダイクストラ法の性能を比較する。第 1 列は優先度付きキュー [1] で decrease-key を行うもの、第 2 列は優先度付きキュー [1] で decrease-key を行わないもの、第 3 列は自動的に decrease-key を行う Heap dict[4] を使用するもの、第 4 列は python の heapq[3] である。

5.2 の実験結果からみると、decrease-key を用いたダイクストラ法は大規模グラフの条件で良好な結果を得ることができいる。Python の heapq[3] は decrease-key のような操作を実装していないが、heapq[3] のデータ構築素敵である。

networkX のダイクストラ法は本研究で用いた [1] のダイクストラ法より標準ため、decrease-key を用いた networkX のダイクストラ法が heapq[3] を用いた networkX のダイクストラ法より高速な結果を得ることができると期待する。

5.5.1 実験結果

networkX のダイクストラ法のデータ構造の関係で、あまり大きなグラフは使えないため、頂点数が 100 万以下のデータを実験する。

頂点 0 を始点とし、5 つの実験の結果を平均し、ダイクストラ法の各バージョンについてかかった時間を測定し、異なる実験で比較を行うものである。実行時間 (表 5.7) の単位は秒で表示している。

表 5.7: networkx のダイクストラ法の性能 (実行時間)

頂点数	decrease-key	no-decrease-key	heap dict	heapq
10	0.0002	0.00019	0.000198	0
100	0.0021	0.0015	0.0019	0.001
1,000	0.028	0.036	0.026	0.01
10,000	0.356	0.383	0.32	0.118
100,000	4.97	5.04	3.85	1.343
264,346	10.22	10.88	9.33	6.687
321,270	11.19	11.38	9.802	6.306
435,666	19.6	21	16.85	13.04

5.5.2 結果分析

実行時間 (表 5.7) から分析すると、heapq[3] のバージョンは decrease-key を使わなくても他のバージョンより高速な結果がで出ている。優先度付きキュー [1] の 2 つバージョンの結果を比較すると、頂点数が少ない場合 (100 以下) は、decrease-key なしのほうがよいである。

heapdict[4] の場合に、表 5.7 と表 5.1 を比較すると、頂点数が 1000 以下の場合に、networkx のダイクストラ法が速い。

第6章　まとめ

ダイクストラのアルゴリズムを使用して最短問題 (一対全など) を解く場合、優先度付きキューを使用して探索のための頂点を選択する時間を短縮することで、ダイクストラのアルゴリズムの効率を最適化することができる。その上で、頂点数が少ない (100 以下) 場合を除き、decrease-key 操作を加えることで、優先度付きキューに冗長な情報を格納することを避け、探索頂点の選択の時間効率を最適化し、探索頂点の候補を追加する時間を短縮している。decrease-key を使用することで、優先度付きキューのメモリの変化量を減らすことができるため、アルゴリズムの高速終了を実現することができると考えである。

謝辞

本研究を進めるにあたり、多くの方々のご支援をいただいたことに感謝いたします。このような新冠流行の状況下で、本研究を総合的にご指導いただけた主指導久保幹雄教授には、感謝の念に堪えません。副指導橋本英樹準教授にご指導を頂き、心より感謝いたします。本研究室の皆様には、本研究期間中に疑問が発生した際にご協力をいただき、ありがとうございました。また、ゼミを通じて久保幹雄先生と皆様から多くの知識を学ぶことで深く感謝いたします。

参考文献

[1] Niceola Amadio：“Dijkstra’s Python Implementations:with and without decrease-key”,AY 2019/2020 , https://github.com/amadionix/dijkstra-python

[2] 安井 雄一郎, 藤澤 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・リサーチ学会和文論文誌, Vol. 54, 2011, pp. 58 – 83

[3] heapq, https://docs.python.org/zh-cn/3/library/heapq.html

[4] Daniel Stutzbach ,heap dict , https://github.com/Daniel Stutzbach/heap dict