61908697 佐々木良輔

問 1

ここでは密度行列の Djikstra 法と Bellman-Ford 法を比較した．図 1 に Dijkstra 法でのノード数 N と Update の回数を示す．またそれぞれの系列について $y(x) = ax^n$ という関数で最小二乗 fitting を行った．表 2 に Dense graph の場合と Sparse graph の場合で得られた n を示す．したがってそれぞれの系列で計算量は $O(N^2)$ 程度になっていることがわかる．

また図 2 に Bellman-Ford 法でのノード数 N と Update の回数を示す．これについても同様に $y(x) = ax^n$ で fitting を行い, その結果は表のようになった．したがって Dense graph では概ね $O(N^2)$ の計算量であるのに対し, Sparse graph では計算量が減っていると考えられる．

表 1　Dijkstra 法での各系列の次数

	n
Dense graph	2.00 ± 0.00
Sparse graph	2.00 ± 0.00

表 2　Bellman-Ford 法での各系列の次数

	n
Dense graph	1.99 ± 0.00
Sparse graph	1.41 ± 0.00

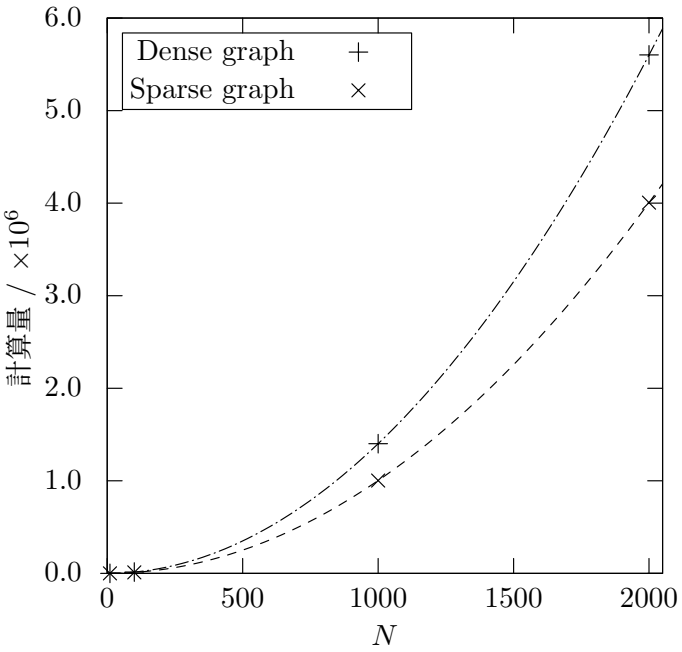

図 1　Dijkstra 法でのノード数と計算量の関係

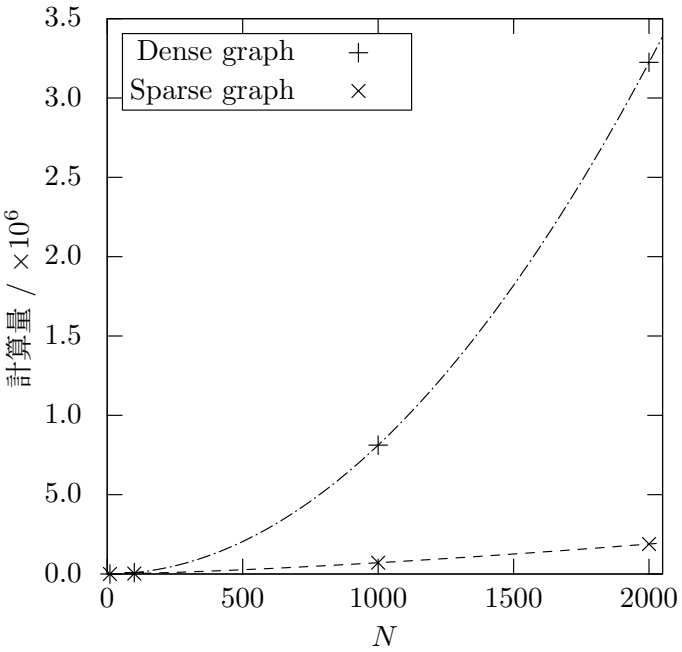

図 2　Bellman-Ford 法でのノード数と計算量の関係

問 2

探索アルゴリズムを用いて原子番号から元素名を探索した．探索アルゴリズムには線形探索と二分探索を用いた．入力データは [1] から取得したものを Excel でランダムに並び替え, データ数を 20, 40, 60, 80, 100, 118 とした．これらのデータ全要素に対して検索を行った際の計算量を比較した. ただし交換操作 1 回はコピー操作 3 回分とした．図 3 に結果を示す. 線形探索は $y(x) = ax + b$, 二分探索は $y(x) = a\log(x)$ という関数で最小二乗 fitting を行った．この結果は線形探査の計算量が $O(N)$ 程度, 二分探索の計算量が $O(\log N)$ 程度であるという事実に整合する．

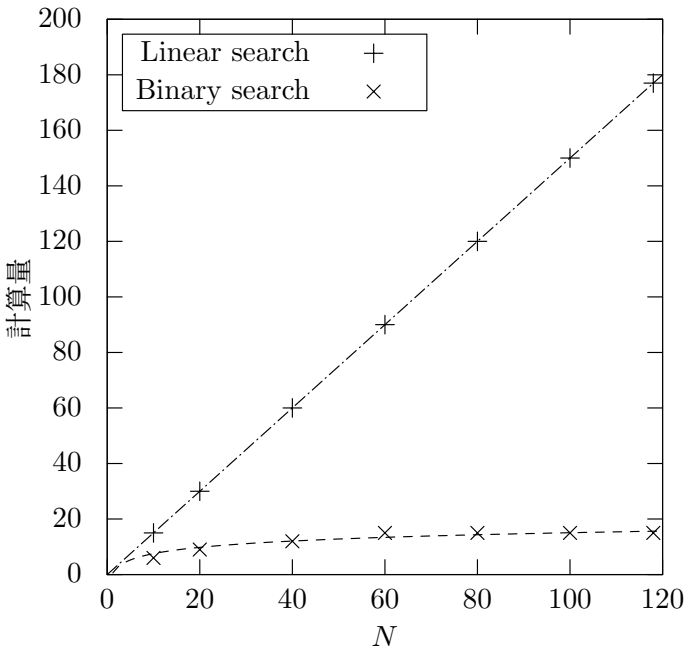

図 3　線形探索と二分探索の比較

問 3

文字列検索のアルゴリズムとして Bitap 法というものがある．他のアルゴリズムと比較して Bitap 法はあいまい検索に対応する, 検索がビット演算のみで行われるため高速であるといった特徴がある.[4]

Bitap 法は検索パターン p から生成されるビットマスクを用いたビット演算により状態遷移図を遷移させることで検索を行う．例として対象文字列 T を abdcababca, 検索パターン p を abca とする. このとき各文字に対するビットマスク m_i は表 5 のようになる．ただし m_0 はパターンに含まれない全ての文字に対応する．このように文字 i に対応するビットマスクはパターンにおいて i が存在するところに 1 が立つ．また状態遷移図としてパターン文字数と同じ長さのビット列 (レジ

スタ R) を用いる. レジスタに対し以下の演算を行うことで検索を行う. ただしレジスタの初期状態は全ビットが 0 である.[4]

1. レジスタを 1 ビット左にシフトする
2. LSB に 1 を立てる
3. 現在位置の文字に対応するマスクを AND 演算する
4. 次の文字に進み同じ演算を繰り返す.

これにしたがって演算をした場合のレジスタの遷移を表 4 に示す. 表 4 のようにレジスタの n 番目に 1 が立っているとき, 現在の検索位置において p と T が n 文字目まで一致している. すなわち MSB に 1 が立ったときパターンが一致している.

また p が”ab.a”のようにあいまい部分を含んでいるときビットマスクを表のようにすることで検索を実行できる.

表 3　ビットマスク

$m_i \backslash p$	a	b	c	a
m_a	1	0	0	1
m_b	0	1	0	0
m_c	0	0	1	0
m_0	0	0	0	0

表 4　状態遷移

		入力文字列									
	初期状態	a	b	d	c	a	b	a	b	c	a
R	0	1	0	0	0	1	0	1	0	0	0
	0	0	1	0	0	0	1	0	1	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	1

表 5　ビットマスク

$m_i \backslash p$	a	b	.	a
m_a	1	0	1	1
m_b	0	1	1	0
m_0	0	0	1	0

問 4

OpenMP を用いて Quick sort を並列化しスレッド数と計算時間の関係を調べた. ソースコードを Listing 1 に示す. 再帰呼出しが 5 回行われるまでは新規タスクを生成し, それ以上の深さでは逐次的に計算した. また計算を実行した計算機の諸元を表に示す. 図 4 にスレッド数と計算時間の関係を示す. 計算時間はスレッド数が 1, 2, 4, 6 それぞれの場合で 10 回計算を行い, その平均を取った. ここで全計算量に対して並列処理が可能な部分の割合を r, スレッド数を N とすると計算時間 T は $T \propto r/N + (1-r)$ となると考えられるので, 計測された計算時間を $y(x) = a(r/x + (1-r))$ という関数で最小二乗 fitting を行い図 4 の点線を得た. fitting により $r = 0.604 \pm 0.076$ を得た. すなわち並列化可能部分の割合が 6 割程度であると考えられる.

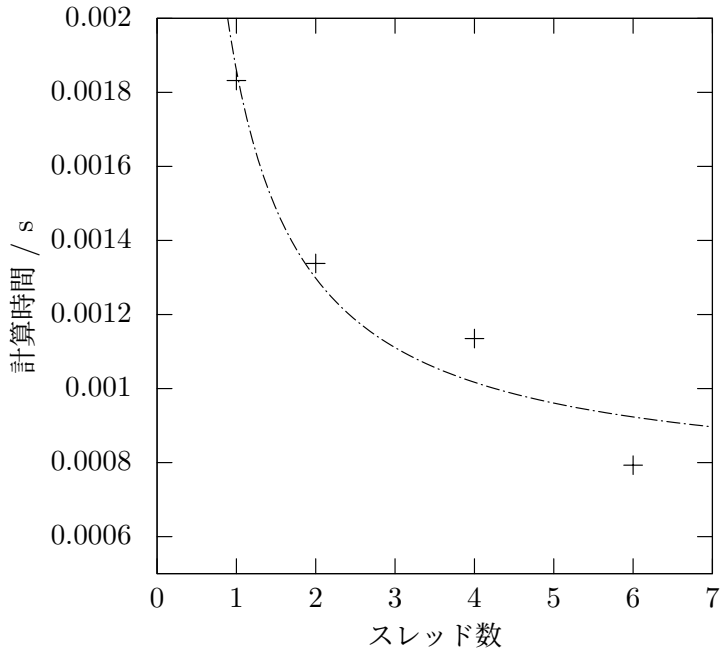

図 4　計算時間とスレッド数の関係

表 6　計算機の諸元

	内容
CPU	intel Xeon E5-2690 v4 (2.6 GHz, 仮想マシンへの割当は 6 コア)
OS	Ubuntu 20.04 LTS
RAM	115.38 GB (/proc/meminfo から)
他	Azure lab service 上のインスタンス

問 5

c++ で Aho-Corasick 法を実装し, 円周率から任意の数列を検索した. ソースコードを Listing 2 に示す. ソースコードは [3] を参考に実装した. 入力テキストとして [2] から円周率 100 万桁を取得し, 整形したものを用いた. また, 検索パターンを 11111, 22222, ..., 99999 とし, 検索パターンの個数を 1 から 9 個に変化させながら検索に要した時間を計測した. 図 5 にその結果を示す.

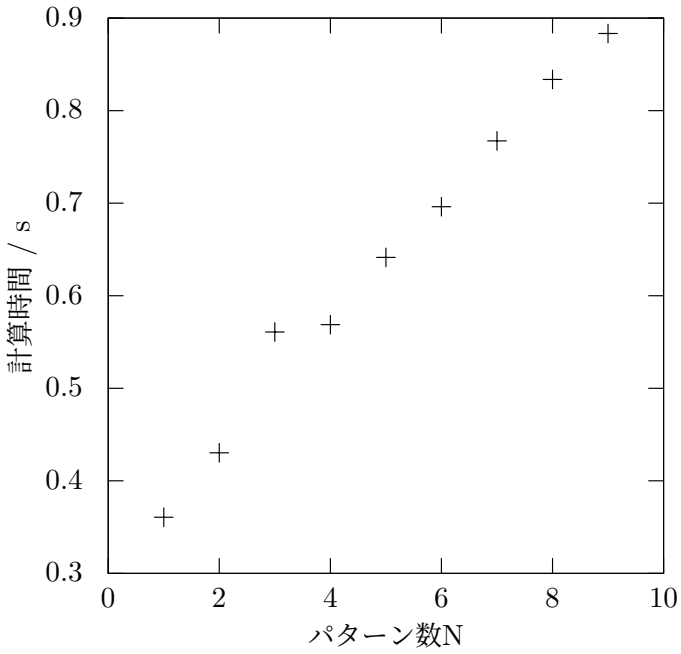

図 5　検索パターン数と検索時間

参考文献

[1] Periodic table of elements.csv. https://gist.github.com/GoodmanSciences/c2dd862cd38f21b0ad36b8f96b4bf1ee. (Accessed on 05/25/2021).

[2] 円周率 (pi) 100 万 (1,000,000) 桁 - ts テクノロジー. https://www.tstcl.jp/ja/randd/pi.php. (Accessed on 05/27/2021).

[3] Aho corasick 法 - naoya のはてなダイアリー. https://naoya-2.hatenadiary.org/entry/20090405/aho_corasick, 04 2009. (Accessed on 05/27/2021).

[4] Weblio 辞書. Bitap アルゴリズム - bitap アルゴリズムの概要 - weblio 辞書. https://www.weblio.jp/wkpja/content/Bitap%E3%82%A2%E3%83%AB%E3%82%B4%E3%83%AA%E3%82%BA%E3%83%A0_Bitap%E3%82%A2%E3%83%AB%E3%82%B4%E3%83%AA%E3%82%BA%E3%83%A0%E3%81%AE%E6%A6%82%E8%A6%81, 09 2020. (Accessed on 05/26/2021).

A 付録

Listing 1 Quick sort

```
#define depth_thresh 5

void quick_func(int left, int right, int depth) {
        int i, j;
        struct student pivot;
        if (left >= right) return;
        i = left;
        j = right;
        my_copy(&table[(i + j) / 2], &pivot);
        do {
                while (my_compare(&table[i], &pivot)==-1) i++;
                while (my_compare(&table[j], &pivot)==1) j--;
                if(i <= j) {
                        my_swap(&table[i], &table[j]);
                        i++;
                        j--;
                }
        } while (i <= j);
        if (depth < depth_thresh) {
                #pragma omp task
                quick_func(left, j, depth + 1);
                #pragma omp task
```

```
23                 quick_func(i, right, depth + 1);
24                 #pragma omp taskwait
25         } else {
26                 quick_func(left, j, depth + 1);
27                 quick_func(i, right, depth + 1);
28         }
29 }
30
31 void Quick(int size) {
32         struct timeval stime, etime;
33         double sec;
34         gettimeofday(&stime, NULL);
35
36         #pragma omp parallel
37         {
38                 #pragma omp single
39                 quick_func(0, size - 1, 0);
40         }
41
42         gettimeofday(&etime, NULL);
43         sec = (etime.tv_sec - stime.tv_sec) +
44                         (etime.tv_usec - stime.tv_usec) / 1000000.0;
45         printf("Elapsed Time %.4f [sec] \n", sec);
46 }
```

Listing 2 Aho-Corasick

```
#include <unordered_map>
#include <vector>
#include <queue>
#include <iostream>
#include <fstream>
#include <string>
#include <sys/time.h>

typedef struct StateMachine
{
    private:
        const int _id;
    public:
        std::unordered_map<char, StateMachine*> next;
        StateMachine(int id) : _id(id){}
        bool has_key(const char &x){
            if(this->next.find(x)==this->next.end()) return false;
            return true;
        }
        void add_next(const char &x, StateMachine* ptr){ next[x] = ptr; }
        StateMachine* get_next(const char &x){ return next[x]; }
        int get_id(){ return _id; }
} _STATEMACHINE;

class MachineAC
{
    public:
        MachineAC(std::vector<std::string> &patterns){
            state.push_back(new StateMachine(0));
            make_goto(patterns);
            make_failure();
        }
        ~MachineAC(){ for(int i = 0; i < state.size(); i++) delete state[i];
             }
        void match(std::string &text){
            StateMachine *s = state[0];
            for(int i = 0; i < text.size(); i++){
                while(go(s, text[i]) == 0) s = failure[s->get_id()];
                s = go(s, text[i]);
                for(std::string found : output[s])
                    std::cout << found << "\tfrom " << i-found.size()+1 << "
```

```
                        to " << i << std::endl;;
41          }
42      }
43
44  private:
45      std::vector<StateMachine*> state;
46      std::vector<StateMachine*> failure;
47      std::unordered_map<StateMachine*, std::vector<std::string>> output;
48      void make_goto(std::vector<std::string> &patterns){
49          for(std::string pattern : patterns){
50              StateMachine *current = state[0];
51              for(char& p : pattern){
52                  if(!current->has_key(p)){
53                      StateMachine *next = new StateMachine(state.size());
54                      current->add_next(p, next);
55                      state.push_back(next);
56                  }
57                  current = current->get_next(p);
58              }
59              output[current].push_back(pattern);
60          }
61      }
62      void make_failure(){
63          failure.resize(this->state.size());
64          std::fill(failure.begin(), failure.end(), this->state[0]);
65          std::queue<StateMachine*> queue;
66          StateMachine *state;
67          StateMachine *next;
68          StateMachine *f;
69          queue.push(this->state[0]);
70          while(queue.size()){
71              state = queue.front();
72              for(auto itr = state->next.begin(); itr != state->next.end();
                     itr++){
73                  next = go(state, itr->first);
74                  if(next != 0) queue.push(next);
75                  if(state != this->state[0]){
76                      f = failure[state->get_id()];
77                      while(go(f, itr->first) == 0) f = failure[f->get_id()
                            ];
78                      failure[next->get_id()] = go(f, itr->first);
79                      output[next].insert(output[next].end(),
```

```
80                              output[failure[next->get_id()]].begin(), output[
                                    failure[next->get_id()]].end());
81                      }
82                  }
83                  queue.pop();
84              }
85          }
86          StateMachine* go(StateMachine *state, const char &x){
87              if(state->has_key(x)){
88                  return state->get_next(x);
89              }else if(state->get_id() == 0){
90                  return state;
91              }else{
92                  return 0;
93              }
94          }
95  };
96
97  int main(void)
98  {
99          std::vector<std::string> patterns{"123456"};
100         MachineAC machine(patterns);
101         std::ifstream ifs("./pi.dat");
102         std::string text;
103         struct timeval stime, etime;
104         double sec;
105
106         if(ifs.fail()){
107                 std::cout << "Failed opening file" << std::endl;
108                 return -1;
109         }
110         getline(ifs, text);
111         gettimeofday(&stime, NULL);
112
113         machine.match(text);
114
115         gettimeofday(&etime, NULL);
116         sec = (etime.tv_sec - stime.tv_sec) +
117                         (etime.tv_usec - stime.tv_usec) / 1000000.0;
118         printf("Elapsed Time %.4f [sec] \n", sec);
119 }
```