پروژهٔ نهایی درس ساختمان داده ها
استاد: دکتر آبین

طراحی و تست عملکرد یک سیستم پیشنهاد دهندهٔ آنلاین مبتنی بر ساختار دادهٔ درخت BJR

نویسنده‌ی اول
مهدیار صلواتی
کد دانشجویی: ۴۰۲۲۴۳۰۸۰
ma.salavati@mail.sbu.ac.ir

نویسنده‌ی دوم
حامی جهانیان
کد دانشجویی: ۴۰۲۲۴۳۱۱۹
haami.jahanian@mail.sbu.ac.ir

چکیده

در این پروژه قصد داریم برای پیدا کردن بهترین پیشنهاد از ساختمان دادهٔ درخت BJR استفاده کنیم. این ساختمان داده به ما اجازه می دهد تا بتوانیم با پردازش تعداد زیادی از نقاط بتوانیم آنهایی را که نقاط دیگر را dominate می کند پیدا کرده و به عنوان نقاط skyline و بهینه خروجی دهیم.

۱.۰ روند کلی

در سه بخش اصلی سعی داریم برای مسئلهٔ مطرح شده، سه نوع پاسخ متفاوت و بهینه بر اساس درخت BJR ارائه دهیم.

در نخستین بخش توابع inject و eject را بر اساس آنچه در صورت پروژه مطرح شده پیاده می کنیم و بر اساس timestep های تعریف شده، نقاط را با استفاده از تابع inject فعال و با eject غیرفعال می کنیم. با توجه به آنکه origin تمام نقاط را dominate می کند، در عمل چنین نقطه ای نمی تواند پاسخ ما باشد. در نتیجه این فرزندان origin خواهند بود که به عنوان بهترین پاسخ ها و به اصطلاح skyline معرفی می شوند.

در بخش دوم قصد داریم با ایجاد تغییراتی در تابع inject با استفاده از الگوریتم lazy evaluation درخت را در هر timestep به صورت balance حفظ کنیم. این موضوع ما را قادر می سازد تا هنگامی که با تعداد گره از مرتبهٔ $\mathcal{O}(n)$ طرف هستیم بتوانیم با انجام $\mathcal{O}(\log(n))$ مقایسه عضو جدید را جانمایی کنیم.

در بخش سوم برای جلوگیری از محاسبات تکراری از یک cache با پیچیدگی حافظهٔ $\mathcal{O}(n)$ استفاده خواهیم کرد.

در انتها علاوه بر صحت سنجی کد ارائه شده، مقایسه ای از سه روش بیان شده انجام خواهیم داد که می تواند برای مطالعات آتی کلیدی باشد.

فهرست مطالب

فصل ۱

پیش نیازها

۱.۱ ساختار دادهٔ لیست پویا

در تمامی بخش ها نیازمند به ساختار داده ای هستیم که عملیات ها روی آن صورت بگیرد. از آنجا که حجم ورودی های تست ها نسبتا بالاست از لیست پویا کمک می گیریم و پیاده سازی دستی این امکان را به ما می دهد تا با دستی باز تر حافظه را مدیریت کنیم.

سه فیلد خصوصی برای نگه داشتن آرایه، ظرفیت کلی و ظرفیت فعلی در نظر می گیریم:

```
private:
    T *data = nullptr;
    size_t capacity = 0;
    size_t size = 0;
```

دو نوع کانستراکتور پیاده می کنیم که یکی از آنها فضای ابتدایی مشخصی را می تواند دریافت کند. هرچند با پر شدن آن لیست دوباره فرایند allocation را انجام خواهد داد:

```
    CustomList() {}

    CustomList(size_t initialCapacity) {
        capacity = initialCapacity;
        size = 0;
        data = new T[capacity];
    }
```

برای انجام deepcopy با مدیریت حافظه متد زیر را تعریف می کنیم:

```
    CustomList &operator=(const CustomList &other) {
        if (this != &other) {
            delete[] data;
```

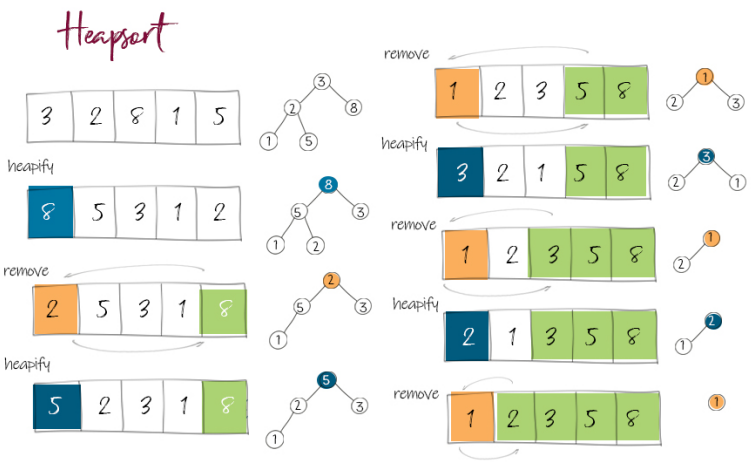
```
            capacity = other.capacity;
            size = other.size;
            data = new T[capacity];
            for (size_t i = 0; i < size; i++) {
                data[i] = other.data[i];
            }
        }
        return *this;
    }
```

از آنجا که در درخت BJR ترتیب آمدن فرزندان ریشه اهمیتی ندارد،می توان در راستای حذف یک عنصر از لیست صرفا آن را با عنصر آخر لیست جایگزین کرده و size را یکی کم کنیم. این گونه دیگر نیاز به شیفت اعضا با پیچیدگی $\mathcal{O}(n)$ نداریم و با $\mathcal{O}(1)$ عملیات حذف را انجام می دهیم:

```
    void removeByIndex(size_t index) {
        if (index >= size) return;
        if (index != size - 1) {
            data[index] = data[size - 1];
        }
        size--;
    }
```

۲.۱ مرتب سازی heapsort

بر اساس آنچه در درس آموختیم و شکل زیر به وضوح نشان می دهد می توانیم این مرتب سازی را تعریف کنیم:




```
    void heapify(CustomList<int> &arr, int n, int i) {
    int largest = i;
```

```
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(CustomList<int> &arr) {
    int n = arr.getSize();
    if (n <= 1) return;

    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

۳.۱ تعریف گره

هر گره باید یک مشخصه (index) که همان شماره خطی است که در آن تعریف شده است داشته باشد و همچنین دارای یک والد و تعدادی فرزند است:

```
    struct node {
    int index = -1;
    node *parent = nullptr;
    CustomList<node *> children;

    node() {
        children.reserve(4);
    }
```

```
};
```

فصل ۲

پیاده سازی توابع مربوط به درخت

۱.۲ تابع dominate

این تابع که می توان گفت بیشترین فراخوانی را نسبت به دیگر توابع دارد، بررسی می کند که آیا یک نقطه دیگری را dominate می کند یا خیر.

دو حالت وجود دارد: - نقطه اول از نقطه دوم بهینه تر باشد: در این حالت نقطه اول نقطه دوم را dominate نمی کند

- نقطه اول، نقطه دوم را dominate نمی کند و همچنین نقطه دوم نیز نقطه اول را dominate نمی کند: برای مثال می توان در خرید خانه این مثال را زد که دو خانه وجود دارند که هر دو هم فاصلهٔ کمی از دریا دارند و هم قیمت مناسبی دارند اما فاصله یکی از دیگری اندکی کمتر اما قیمتش هم اندکی بیشتر است. در این حالت هیچ کدام بر دیگری برتری ندارد.

از آنجا که برخی فیلد ها با بیشتر بودن ارزشمند می شوند (مثل مساحت خانه) و برخی با کمتر بودن ارزشمند می شوند (مانند فاصلهٔ خانه تا دریا) در نتیجه یک Effective List شامل ۱ یا -۱ به این تابع پاس می دهیم تا ملاک ارزشمندی مشخص شود. (باید اشاره شود که در تست های ارائه شده تمامی فیلدها با کمتر بودن ارزشمند می شود پس این را به عنوان ۱ قرار می دهیم):

```
inline bool dominates(int a, int b, const CustomList<int> &efflist) {
bool hasEfflist = !efflist.empty();
bool result = false;

for (int i = 0; i < dim; i++) {
    int eff = hasEfflist ? efflist[i] : 1;

    if (eff > 0) {
        if (dataset[a][i] > dataset[b][i]) return false;
        else if (dataset[a][i] < dataset[b][i]) result = true;
    } else if (eff < 0) {
        if (dataset[a][i] < dataset[b][i]) return false;
        else if (dataset[a][i] > dataset[b][i]) result = true;
```

```
        }
    }
    return result;
}
```

۲.۲ تابع inject

با توجه به الگوریتم تابع را تعریف می کنیم. شایان ذکر است این الگوریتم درخت را طوری تشکیل می دهد که هر چه به ریشه نزدیک تر می شویم likelihood آن که پوینت در مقطعی به عنوان skyline معرفی شود بیشتر می شود. یعنی برگ های این درخت (که دور ترین گره ها از ریشه هستند) گره های دیگر را dominate نمی کنند:

```
    void inject(node *parent, node *newNode) {
    CustomList<node *> childrenL = parent->children;
    size_t numOfChild = childrenL.getSize();
    for (size_t i = 0; i < numOfChild; i++) {
        node *child = childrenL[i];
        if (dominates(child->index, newNode->index, CustomList<int>())) {
            inject(child, newNode);
            return;
        }
    }
    newNode->parent = parent;
    parent->children.insert(newNode);
    CustomList<node *> toMove;
    toMove.reserve(numOfChild);

    for (int i = parent->children.getSize() - 1; i >= 0; i--) {
        node *nodeToRem = parent->children[i];
        if (nodeToRem != newNode && dominates(newNode->index, nodeToRem->
            index, CustomList<int>())) {
            toMove.insert(nodeToRem);
            parent->children.removeByIndex(i);
        }
    }

    size_t toMoveCount = toMove.getSize();
    for (size_t i = 0; i < toMoveCount; i++) {
        node *sibling = toMove[i];
        sibling->parent = newNode;
        newNode->children.insert(sibling);
    }
```

```
}
```

۳.۲ تابع eject

این تابع گره را حذف کرده و جانمایی گره هایی که نیازمند به تغییر موقعیت شده اند را با بهره گیری از تابع inject انجام می دهد:

```
    void eject(node *nodeToRem) {
    node *parent = nodeToRem->parent;
    CustomList<node *> childrenL = nodeToRem->children;

    parent->children.removeItem(nodeToRem);

    CustomList<node *> childrenCopy = nodeToRem->children;
    size_t numOfChild = childrenCopy.getSize();
    for (size_t i = 0; i < numOfChild; i++) {
        node *child = childrenCopy[i];
        inject(parent, child);
    }

    delete nodeToRem;
}
```

۴.۲ توابع مربوط به کار با فایل

۵.۲ خواندن نقاط از روی فایل

نقاط را از فایل خوانده و آن را در یک ماتریس قرار می دهیم. بنابراین ستون های این ماتریس نمایان گر feature ها و ردیف های آن نمایان گر index نقاط خواهند بود.

برای جلوگیری از allocation های پیاپی که overhead قابل توجهی دارد، یک فضای بزرگ برای سطر های این ماتریس قرار می دهیم و برای ستون های آن ۷ درایه کافی است چرا که بزرگترین تست ما حاوی ۷ ویژگی است:

```
    CustomList<CustomList<int>> readMatrixFromFile(const string &filename
        ) {
    CustomList<CustomList<int>> matrix;
    ifstream file(filename);

    matrix.reserve(100000);

    string line;
```

```
    line.reserve(256);

    while (getline(file, line)) {
        CustomList<int> row;
        stringstream ss(line);
        int num;

        row.reserve(7);

        while (ss >> num) {
            row.insert(num);
        }

        if (!row.empty()) {
            matrix.insert(row);
        }
    }

    file.close();
    return matrix;
}
```

۶.۲ خواندن گام های زمانی از فایل

برای این کار دو رویکرد می توان داشت: ۱) در هر زمان، تمام نقاط را بررسی کنیم تا ببینیم چه نقاطی باید عملیات injection یا ejection روی آن ها صورت بگیرد. این گونه این قسمت از کد با پیچیدگی $\mathcal{O}(t \times n)$ که در آن n تعداد کل نقاط است صورت می گیرد که با احتساب اینکه ممکن است n بسیار بزرگ باشد، این راه منطقی به نظر نمی آید.

۲) ماتریس حاصل از فایل را ترانهاده کنیم و دو ماتریس ناکامل از آن نگه داریم. این گونه سطر ماتریس ها نمایان گر زمان start یا زمان end و ستون آنها نمایان گر index آنها خواهند بود. این گونه به راحتی می توان تشخیص داد در زمان t چه تعداد نقاط نیاز به بررسی دارند و تنها کافی است آنها را بررسی کنیم. پس این کار با پیچیدگی $\mathcal{O}(t \times k)$ که k تعداد نقاط فعال در زمان t هستند، قابل انجام است. با توجه به اینکه k در مقایسه با n به شدت کمتر است، راه دوم منطقی تر می تواند باشد و ما همین را پیاده می کنیم:

```
    CustomList<CustomList<CustomList<int>>> readTimesFromFile(const
        string &filename) {
    CustomList<CustomList<int>> starts;
    CustomList<CustomList<int>> ends;
    CustomList<CustomList<CustomList<int>>> result;
    ifstream file(filename);

```

```
    starts.resize(800000);
    ends.resize(800000);


    string line;
    line.reserve(64);

    int pointIndex = 0;
    while (getline(file, line)) {
        stringstream ss(line);
        int start, end;
        if (ss >> start >> end) {
            starts[start].insert(pointIndex);
            ends[end].insert(pointIndex);
            pointIndex++;

            if (end > maxTime) {
                maxTime = end;
            }
        }
    }

    file.close();
    result.insert(starts);
    result.insert(ends);
    return result;
}
```

۷.۲ تابع main

با استفاده از دو تابع اخیر، اطلاعات را از روی فایل دریافت می کنیم. سپس در یک حلقه که بر اساس timestep ها پیمایش می کند با منطقی که ارائه شد، به inject و eject کردن می پردازیم و در انتها فرزندان ریشه را که همان نقاط skyline ما هستند خروجی می دهیم. فقط با توجه به اینکه هر خط فایل خروجی به صورت سورت شده است با استفاده از الگوریتم heapsort که قبل تر راجع به آن صحبت شد، خروجی را ابتدا مرتب کرده و سپس در فایل می نویسیم:

```
    int main() {
    CustomList<CustomList<int>> matrix = readMatrixFromFile("dataset/
        medium/medium.input");
    CustomList<CustomList<CustomList<int>>> times = readTimesFromFile("
        dataset/medium/medium.times");

```

```
    CustomList<CustomList<int>> starts = times[0];
    CustomList<CustomList<int>> ends = times[1];


    dataset = matrix;
    dim = matrix[0].getSize();

    node *origin = new node();

    CustomList<node *> activeNodes(matrix.getSize());
    activeNodes.resize(matrix.getSize(), nullptr);


    ofstream outFile("output.txt");
    outFile.tie(nullptr);

    for (int t = 0; t <= maxTime; t++) {
        size_t startCount = starts[t].getSize();
        for (size_t i = 0; i < startCount; i++) {
            int pointIdx = starts[t][i];
            node *newNode = new node();
            newNode->index = pointIdx;
            inject(origin, newNode);
            activeNodes[pointIdx] = newNode;
        }

        size_t endCount = ends[t].getSize();
        for (size_t i = 0; i < endCount; i++) {
            int pointIdx = ends[t][i];
            if (activeNodes[pointIdx]) {
                eject(activeNodes[pointIdx]);
                activeNodes.removeByIndex(pointIdx);
            }
        }

        CustomList<int> skyline;
        size_t childrenCount = origin->children.getSize();
        skyline.reserve(childrenCount);

        for (size_t i = 0; i < childrenCount; i++) {
```

```
۴۵              node *child = origin->children[i];
۴۶              skyline.insert(child->index);
۴۷          }
۴۸
۴۹          heapSort(skyline);
۵۰
۵۱          size_t skylineSize = skyline.getSize();
۵۲          for (size_t i = 0; i < skylineSize; i++) {
۵۳              if (i > 0) outFile << "␣";
۵۴              outFile << skyline[i];
۵۵          }
۵۶          outFile << "\n";
۵۷      }
۵۸
۵۹      outFile.close();
۶۰      delete origin;
۶۱
۶۲      return 0;
۶۳  }
```

فصل ۳

ساخت درخت با الگوریتم Lazy Evaluation برای Balance نگه داشتن درخت

۱.۳ پیش نیاز ها: توابع depth و desc

تابع depth را به صورت بازگشتی و تا جایی که به ریشه برسیم تعریف می کنیم:

```
    int depth(node *n) {
    if (!n || !n->parent) {
        return 0;
    }
    return 1 + depth(n->parent);
    }
```

تابع desc که قرار است تعداد تمامی نوادگان یک گره را برگرداند، به صورت بازگشتی تعریف می کنیم:

```
int desc(node *n) {
    if (!n) return 0;
    int descendants = 0;
    for (size_t i = 0; i < n->children.getSize(); i++) {
        descendants += 1 + desc(n->children[i]);
    }
    return descendants;
}
```

۲.۳ تابع inject

با استفاده از دو تابع اخیر و همچنین الگوریتم ارائه شده در صورت پروژه این تابع را تعریف می کنیم. دربارهٔ متغیر d می توان گفت که نمایان گر حداکثر ارتفاعی است که درخت می تواند داشته باشد. می توان گفت که هرچه درخت ارتفاع کمتری داشته باشد، تعداد مقایسه ها در آینده برای اضافه کردن گره های بعدی کمتر خواهد بود. این موضوع که کدام d می تواند بهینه باشد را در بخش آنالیز کارایی بیشتر دنبال خواهیم کرد.

```
۱   void inject(node *r, node *v) {
۲   if (r->index == -1 || depth(r) < d) {
۳       CustomList<node *> C = r->children;
۴       int g = INT_MAX;
۵       node *t = nullptr;
۶
۷
۸       size_t childCount = C.getSize();
۹       for (size_t i = 0; i < childCount; i++) {
۱۰          node *c = C[i];
۱۱          if (dominates(c->index, v->index, CustomList<int>())) {
۱۲              int descCount = desc(c);
۱۳              if (descCount < g) {
۱۴                  t = c;
۱۵                  g = descCount;
۱۶              }
۱۷          }
۱۸      }
۱۹
۲۰      if (t != nullptr) {
۲۱          inject(t, v);
۲۲          return;
۲۳      }
۲۴  }
۲۵
۲۶  v->parent = r;
۲۷  r->children.insert(v);
۲۸
۲۹  if (r == nullptr || depth(r) < d) {
۳۰      CustomList<node *> C = r->children;
۳۱
۳۲      for (int i = r->children.getSize() - 1; i >= 0; i--) {
۳۳          node *c = r->children[i];
```

```
            if (c != v && dominates(v->index, c->index, CustomList<int>()
                )) {
                r->children.removeByIndex(i);
                c->parent = v;
                v->children.insert(c);
            }
        }
    }
}
```

سایر توابع بدون تغییر مانند بخش اول باقی می مانند.

فصل ۴

بهره گیری از ND-cache

همان طور که در توضیحات تابع dominate بیان شد، دو نقطه ممکن است هیچ کدام بر دیگری برتری نداشته باشد. از این موضوع که نقاطی که skyline هستند، هیچ کدام دیگری را dominate نمی کنند، می توانیم استفاده کنیم تا با ذخیرهٔ رابطه های غیر مغلوب $(non - dominated\ relation)$ از محاسبات انجام شده در تابع dominate که بارها فراخوانی می شود، جلوگیری کنیم.

این کش پیچیدگی حافظه ای از مرتبهٔ $\mathcal{O}(n)$ دارد و اندیس های آن نمایانگر ایندکس نقاط است و مقادیری که در آن ذخیره می شود، شمارهٔ آخرین timestep ای است که آن نقطه در آن زمان جزو skyline ها قرار داشته است.

برای نمایش بهتر کارکرد این کش، مثال های زیر را می بینیم:

```
NDcache[5] = 10;
```

این یعنی نقطهٔ ۵ اخیرا در T_5 جزو نقاط skyline بوده است.

```
if(NDcache[a] == NDcache[b]) {}
```

قطعه کد بالا چک می کند که آیا نقطهٔ a و نقطهٔ b اخیرا در skyline مشترکی به عنوان skyline حضور داشته اند یا خیر.

با توجه به توضیحات مطرح شده، باید یک لیست به تعداد نقاط بسازیم:

```
CustomList<int> NDcache;
int size = matrix.getSize();
NDcache.resize(size);
```

سپس به عنوان مقادیر اولیه ۱- را مقدار دهی کنیم:

```
for (int i = 0; i < size; i++) {
    NDcache[i] = -1;
}
```

حال هر زمان که نقطه ای را وارد skyline می کنیم، در لیست NDcache هم timestep آن را به روز رسانی می کنیم:

```
    for (size_t i = 0; i < childrenCount; i++) {
            node *child = origin->children[i];
            skyline.insert(child->index);
            NDcache[child->index] = t;
        }
```

در نهایت، در ابتدای تابع dominate از این کش استفاده کرده و چک می کنیم که اگر دو نقطه ای که قصد بررسی آن ها را داریم، اگر اخیرا در یک timestep مشترک جزو skyline بوده اند، پس false خروجی می دهیم چرا که نمی توانند یکدیگر را dominate کنند:

```
    inline bool dominates(int a, int b, const CustomList<int> &efflist) {
    if (NDcache[a] == NDcache[b] and NDcache[a] != -1) {
        return false;
    }
    ...
```

فصل ۵

بررسی صحت عملکرد

کد هایی که تا اینجا نوشتیم همگی در فایل output.txt خروجی خود را وارد می کردند. در نتیجه برای مقایسهٔ این فایل با فایل های با پسوند .refout دیتاست های داده شده از تکه کد سادهٔ زیر بهره می گیریم:

```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    string inputSize = "medium";
    ifstream file1("output.txt");
    ifstream file2("dataset/" + inputSize + "/" + inputSize + ".refout");


    string line1, line2;
    int lineNum = 1;

    while (getline(file1, line1) && getline(file2, line2)) {
        if (line1 != line2) {
            cout << "Files are different at line " << lineNum << endl;
            return 0;
        }
        lineNum++;
    }

    if (getline(file1, line1) || getline(file2, line2)) {
        cout << "Files have different number of lines" << endl;
```

```
            return 0;
        }

        cout << "Files are identical" << endl;
        return 0;
}
```

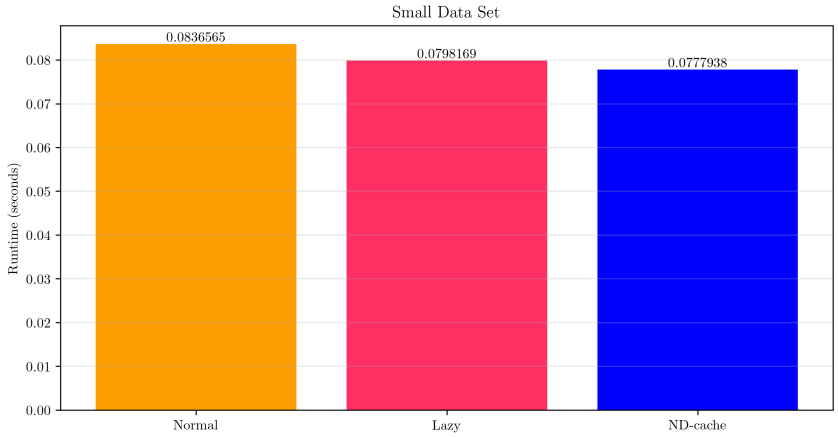
حال به ازای هر سه inputSize مختلف و به ازای سه کد مختلف تست را انجام می دهیم و در همگی با پیام زیر روبرو می شویم:

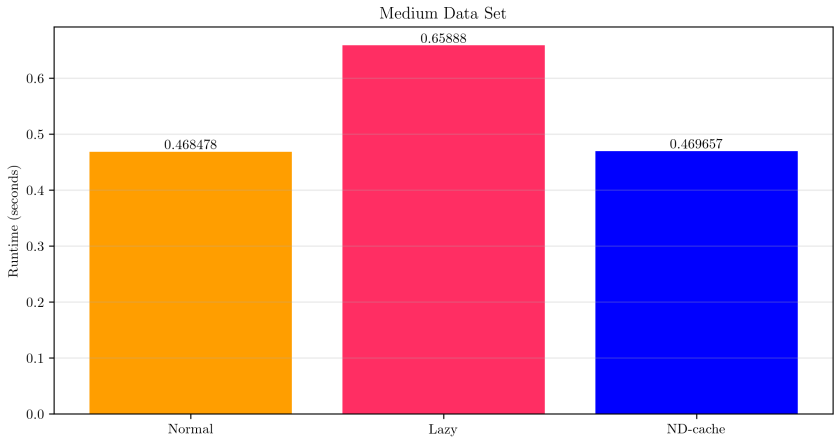
```
Files are identical
```

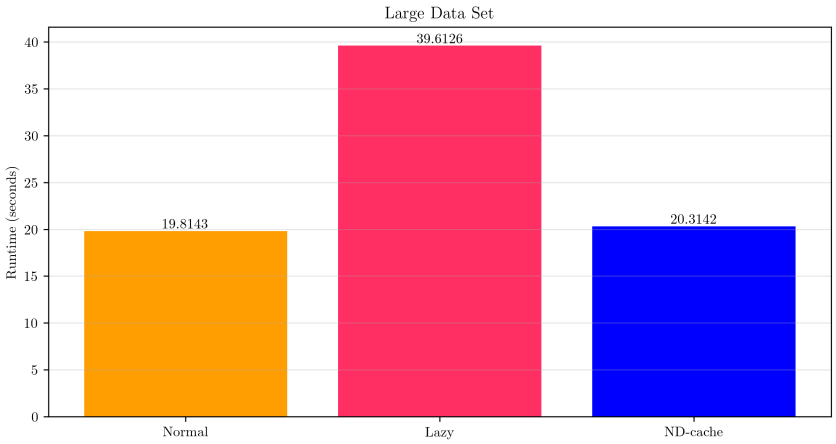
فصل ۶

بررسی کارایی

در این بخش قصد داریم زمان اجرای الگوریتم های مختلفی که در فصل های گذشته دیدیم را با هم مقایسه کنیم. شایان ذکر است تمامی کد ها روی پردازندهٔ $M1$ صورت گرفته و تا حد ممکن سعی شده تا تست ها در شرایط برابر انجام شوند.







به طور کلی می توان دید که روش های Normal و ND-cache بسیار مشابه همدیگر هستند اما روش Lazy Evaluation از سایرین بیشتر به طول می انجامد.

شایان ذکر است در تمامی تست های برای Lazy Evaluation از $d = ۴$ استفاده شده است. چرا که با توجه به نتایج مقاله و همچنین تست های خودمان این مقدار بهترین کارایی را دارد.

الگوی داده های ورودی بسیار می توانند در این مقایسه ها تاثیر گذار باشند و نمی توان از این نتایج یک قاعدهٔ کلی بدست آورد. برای مثال روش Lazy Evaluation همانطور که گفتیم باعث می شود ارتفاع درخت در مراحل مختلف از مرتبهٔ $\mathcal{O}(\log(n))$ باشد و این سبب می شود تا برای درختانی که تعداد گره زیادی دارند، به کارایی مطلوبی برسیم. اما در دیتا ست داده شده بیشترین تعداد گره در یک timestep واحد ۱۰۳۶۴ تا است و شاید به این دلیل است که به کارایی مطلوب نرسیدیم.