



پروژه‌ی نهایی درس ساختمان داده‌ها

پیاده‌سازی درخت BJR برای حل مسئله‌ی Skyline

نازنین فروتن ۴۰۲۲۴۳۰۸۴

ریحانه داداش‌پور ۴۰۲۲۴۳۰۶۰

| مقدمه

- تعریف مسئله: در این پروژه با استفاده از درخت BJR و کش ND، هدف محاسبه مجموعه نقاط Skyline از یک مجموعه داده است. نقاط Skyline نقاطی هستند که در هیچ بعدی توسط هیچ نقطه دیگری کاملاً تسلط ندارند. این نقاط در مسائل مختلفی مانند پایگاه‌های داده، تصمیم‌گیری، و تحلیل داده‌ها کاربرد دارند.
- هدف پروژه: بررسی روش‌های بهینه‌سازی برای محاسبه Skyline در فضای چندبعدی با استفاده از BJR-tree و ND-Cache.

| توضیح مسأله

○ عناصر

- Skyline: مجموعه‌ای از نقاط است که هیچ نقطه‌ای در تمام ابعاد بر آن تسلط ندارد. این نقاط، برای تحلیل‌های مختلف، معمولاً اطلاعات بیشتری نسبت به دیگر نقاط دارند.
- فضای چندبعدی: با افزایش ابعاد، مسئله‌ی Skyline پیچیده‌تر می‌شود. تعداد نقاط Skyline ممکن است به‌طور غیرمنتظره‌ای زیاد شود، مخصوصاً زمانی که ابعاد فضای داده زیاد باشد.
- مفهوم تسلط (Domination): نقطه‌ای a ، نقطه‌ای b را Dominate می‌کند اگر در تمامی ابعاد، a بهتر از b باشد.
- Continuous Skyline Computation: اگر مجموعه داده پویا باشد (یعنی نقاط به‌طور پیوسته وارد و خارج شوند)، نیاز به محاسبه‌ی پیوسته نقاط Skyline خواهیم داشت.

○ ساختار داده

BJR-Tree (a)

- ساختار درخت BJR: در درخت BJR، هر گره نماینده یک نقطه است و یال‌ها نشان‌دهنده روابط تسلط هستند. اگر یالی از a به b وصل باشد، به این معناست که a نقطه b را تسلط می‌کند.

- ویژگی‌های **BJR-Tree** :

- درخت سلسله‌مراتبی است.
- مستقل از ابعاد عمل می‌کند.
- نقاط Skyline به ریشه وصل می‌شوند.
- گره‌هایی که به ریشه نزدیک‌ترند، Skyline potential بیشتری دارند.

- **Lazy Strategy**: این استراتژی در مقایسه با روش معمول که در آن به فرزندان ریشه بررسی می‌شود، به تعداد اولاد هر گره توجه می‌کند و به طور هوشمندتر گره‌ها را بررسی می‌کند تا طول درخت را کاهش دهد.

ND-Cache (b)

- **وظیفه‌ی ND-Cache**: این کش اطلاعات non-dominance را نگه می‌دارد تا از محاسبات مجدد جلوگیری کند.
- **روش کار**: هر زمان که یک نقطه Skyline باشد، اطلاعات آن در کش ذخیره می‌شود. برای بررسی اینکه آیا نقطه‌ای در زمان‌های مختلف یکدیگر را dominate می‌کنند یا نه، می‌توان از کش استفاده کرد.
- **اطلاعات ذخیره‌شده**: به‌طور مثال، اگر نقطه‌ای در زمان T3 جزء Skyline باشد، در کش ذخیره می‌شود که در زمان T3، این نقطه Skyline بوده است.

Continuous Skyline Computation (c)

- **فعال‌سازی و غیرفعال‌سازی نقاط**: نقطه‌ای می‌تواند فعال یا غیرفعال شود بسته به اینکه در بازه زمانی فعلی جزء Skyline است یا خیر.
- **Skyline Potential**: پتانسیل یک نقطه برای Skyline بودن به تعداد نقاطی که آن را تسلط می‌کنند بستگی دارد. هرچه تعداد نقاط تسلط بیشتر باشد، پتانسیل آن برای Skyline شدن کمتر است.

| پیاده‌سازی

برای پیاده‌سازی این پروژه از دو فایل استفاده شده که شامل `main.cpp` و `modules.hpp` می‌شه. فایل `main` شامل توابع مربوط به ران شدن و خواندن فایل‌ها و تست کیس است. فایل `modules` هم شامل کلاس‌هایی هستند که برای ساخت درخت و کار با BJR tree به‌اشون کار داریم.

• `modules.hpp`

○ ساختار `point`

در فایل `modules.hpp`، ساختار `Point` به‌طور مستقیم برای نگهداری اطلاعات هر نقطه از مجموعه داده‌ها استفاده می‌شود. هر نقطه دارای شناسه (`id`) و مقدارهایی برای هر بعد است که در آرایه `values` ذخیره می‌شود.

```
1 struct Point {
2     int id;
3     int dim;
4     int* values;
5
6     Point() : id(-1), dim(0), values(nullptr) {}
7     Point(int id, int dim) : id(id), dim(dim) {
8         values = new int[dim];
9     }
10 };
11
```

○ ساختار `node`

درخت ما از گره‌ها (`Node`) تشکیل می‌شود. هر گره نماینده یک نقطه است و به فرزندان خود لینک دارد. اگر گره‌ای به ریشه متصل باشد، یعنی آن نقطه `Skyline` است.

Data Structures Project Report

```
1 struct Node {
2     Node* head_child;
3     Node* next;
4     Node* parent;
5     Point point;
6     bool is_root = false;
7
8     void add_child(Node* child) { }
9
10    int children_size(){ }
11
12    void remove_child(Node* child){ }
13
14 };
```

○ ساختار BJR Tree

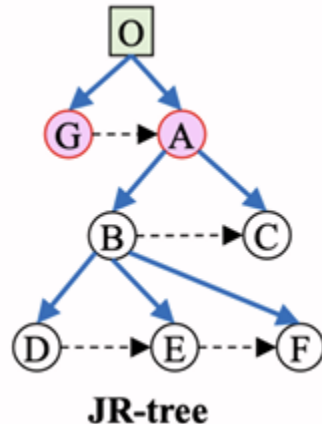
درخت BJR برای ذخیره‌سازی روابط تسلط (Domination) بین نقاط استفاده می‌شود. در این درخت، هر گره یک نقطه است و یال‌ها نشان‌دهنده تسلط هستند.

```
1 struct BJR_tree{
2     Node* root;
3     int total_points = 0;
4     bool* exists;
5     int depth = 0;
6     bool lazy = false;
7     int* ND_cache = nullptr;
8     bool ND_use = false;
9
10    BJR_tree(int num_points) : total_points(num_points) {
11        exists = new bool[total_points];
12        for(int i=0 ; i<total_points ; i++){
13            exists[i] = false;
14        }
15    }
16 }
```

```
16
17     bool does_point_exist(int id){}
18     Node* find_node_by_id(int id, Node* root) {}
19     int Depth(Node* node){}
20     int Desc(Node* node){}
21     void add_to_exists(Node* node){}
22     void remove_exists(Node* node){}
23     void inject(Node* root, Node* new_node){}
24     void lazy_inject(Node* root, Node* new_node){}
25     void eject(Node* example_node) {}
26
27 };
28
```

○ نحوه ذخیره‌سازی دیتا در درخت

- در این نحوه ذخیره‌سازی از روشی مشابه ایده صورت پروژه استفاده شده.
- داده‌ها در ساختار درختی به‌طور خاص به روش JR-tree ذخیره می‌شوند. در اینجا سه ستون اصلی از جمله "آدرس (Address)", "والد (Parent)", "بعدی (Next)" و "سرچکیده (Head child)" برای هر گره در درخت مشخص شده است. این روش ذخیره‌سازی اطلاعات از سه جدول مختلف استفاده می‌کند:
۱. **آدرس (Address):** این ستون به هر گره در درخت یک آدرس خاص اختصاص می‌دهد. در این ساختار، هر گره در درخت یک شناسه منحصر به فرد دارد که آن را از دیگر گره‌ها متمایز می‌کند.
 ۲. **والد (Parent):** این ستون نشان می‌دهد که هر گره به کدام گره دیگر به‌عنوان والد تعلق دارد. مثلاً، گره A والد گره‌های B و C است، و گره B والد گره‌های D، E و F است.
 ۳. **بعدی (Next):** این ستون روابط بین گره‌ها را مشخص می‌کند. اگر گره‌ای فرزند دیگری داشته باشد، به گره بعدی اشاره می‌کند. در برخی موارد، این مقدار "Null" است که نشان‌دهنده این است که گره دیگر فرزند ندارد.
 ۴. **سرچکیده (Head child):** این ستون نشان می‌دهد که هر گره فرزند اول خود را در کدام آدرس ذخیره کرده است. اگر گره‌ای فرزند نداشته باشد، مقدار آن "Null" خواهد بود.



Contents of tree-structure memory

Address	Parent	Next	Head child
A	Root	Null	B
B	A	C	D
C	A	Null	Null
D	B	E	Null
E	B	F	Null
F	B	Null	Null
G	Root	A	Null

○ معرفی متدهای اصلی درخت

Depth() ✓

- توضیح: این تابع معمولاً برای محاسبه عمق درخت یا گره‌ها استفاده می‌شود.
- کاربرد: در پروژه‌های درختی مانند BJR-tree، عمق درخت می‌تواند بر عملکرد الگوریتم تاثیر بگذارد. برای مثال، درخت‌های کم‌عمق معمولاً سریع‌تر عمل می‌کنند. این تابع معمولاً عمق گره را نسبت به ریشه درخت محاسبه می‌کند.



```

1  int Depth(Node* node){
2      int res = 0;
3      Node* current = node;
4      while(current->parent != nullptr){
5          res++;
6          current = current->parent;
7      }
8      return res;
9  }
```

Desc() ✓

- توضیح: این متد برای برگرداندن تعداد نوادگان یک گره استفاده می‌شود.

- کاربرد: برای زمانی استفاده می‌شه که بخوایم از lazy evaluation استفاده کنیم.

```
1  int Desc(Node* node){
2      if(node->head_child == nullptr) return 0;
3
4      int res = 1;
5      Node* current = node->head_child;
6      while(current != nullptr){
7          res++;
8          res += Desc(current);
9          current = current->next;
10     }
11
12     return res;
13 }
```

inject() ✓

- توضیح: این تابع برای وارد کردن یک نقطه جدید به درخت و برقراری روابط تسلط (domination) استفاده می‌شود.
- کاربرد: در درخت BJR، زمانی که یک نقطه جدید به درخت افزوده می‌شود، این تابع باید روابط تسلط را بررسی کرده و نقطه جدید را در مکان مناسب در درخت قرار دهد. در صورت نیاز، باید گره‌ها و یال‌ها به‌طور داینامیک تغییر یابند.

Algorithm 1 New node injection

```
1: procedure inject(parent, e: new node)
2: children  $\leftarrow$  children of parent;
3: for all c  $\in$  children do
4:   if dominates(c, e) then
5:     inject(c, e);
6:   return
7: add e to parent as a child;
8: for all c  $\in$  children do
9:   if dominates(e, c) then
10:    move c to e as a child;
```

lazy_inject() ✓

- توضیح: این تابع مشابه به inject() است، اما با یک تفاوت کلیدی. در این روش، گره‌ها فقط در صورت نیاز به‌طور کامل وارد درخت می‌شوند، یعنی عملیات کمتری انجام می‌دهند و این به‌ویژه در درخت‌های بزرگ که ممکن است به عملکرد آسیب بزنند، مفید است.
- کاربرد: این تابع می‌تواند به‌ویژه در پروژه‌هایی که نیاز به محاسبات سریع‌تر دارند، مفید باشد. این روش باعث می‌شود که از اضافه کردن گره‌ها به‌طور پیوسته جلوگیری شود و فقط زمانی که نیاز است، تغییرات اعمال شود.

eject() ✓

- توضیح: این تابع برای حذف یک نقطه از درخت استفاده می‌شود. زمانی که یک نقطه دیگر جزء نقاط Skyline نیست، باید از درخت حذف شود.
- کاربرد: در درخت‌های دینامیک مانند BJR-tree، زمانی که یک نقطه دیگر شرایط لازم برای Skyline بودن را ندارد (مثلاً توسط نقطه دیگری تسلط شده است)، این تابع باید آن را از درخت حذف کند و گره‌ها و روابط تسلط را به‌روزرسانی کند.

Algorithm 2 Node ejection

```
1: procedure eject( $e$ : node)
2:    $parent \leftarrow$  parent of  $e$ ;
3:    $children \leftarrow$  children of  $e$ ;
4:   remove  $e$  from  $parent$ ;
5:   for all  $c \in children$  do
6:     inject( $parent, c$ );
```

• main.cpp

۱. خواندن داده‌های پیکربندی و ورودی‌ها

داده‌های ورودی شامل چند فایل مختلف هستند و ما سه نوع سایز small, medium و large

داریم:

Data Structures Project Report

- **size.setup**: این فایل شامل پیکربندی داده‌ها است، مثل تعداد ابعاد نقاط، تعداد مراحل زمانی (time steps)، و نوع داده‌ها.
- **small.times**: این فایل شامل اطلاعات مربوط به زمان‌های شروع و پایان هر نقطه است. به عبارت دیگر، این فایل نشان می‌دهد که هر نقطه در چه بازه زمانی فعال است.
- **small.input**: این فایل شامل مقادیر هر نقطه است. برای هر نقطه، مقادیر مختلف آن در ابعاد مختلف ذخیره شده است.

۲. پردازش نقاط برای هر زمان

- برای هر گام زمانی (time step)، باید نقاط فعال را بررسی کنیم و اقداماتی مانند اضافه کردن یا حذف کردن آنها را انجام دهیم.
- در حلقه اصلی **for** که بر روی زمان‌ها می‌چرخد، برای هر تایم:
- **مقایسه با تایم‌های مشخص شده در times**: در این مرحله، به‌ازای هر مرحله زمانی، باید ببینیم که هر نقطه آیا در بازه زمانی خاص خودش فعال است یا خیر. این کار با استفاده از داده‌های موجود در فایل **times** انجام می‌شود که شامل محدوده‌های زمانی شروع و پایان هر نقطه است.

```
1  for(int time = 0 ; time < time_steps ; time++){
2      cout << "time: " << time << endl;
3      id = 0;
4      for(int t = 0 ; t < times.size() ; t++) {
5          range = times[t];
6          ss.clear();
7          ss.str(range);
8          ss >> temp;
9          start = stoi(temp);
10         ss >> temp;
11         end = stoi(temp);
12     }
```

○ چک کردن وضعیت هر نقطه: اگر زمان فعلی در بازه زمانی یک نقطه باشد، آن نقطه باید به درخت اضافه شود. در غیر این صورت، اگر نقطه‌ای خارج از این بازه زمانی باشد، باید از درخت حذف شود.

۳. حذف نقاط غیرفعال

اگر نقطه‌ای از بازه زمانی خود خارج شود یا توسط نقطه دیگری تسلط یابد، باید از مجموعه حذف شود:

```
1  else {
2      if (time < start) break;
3      if ((time >= end || time < start) && my_tree->does_point_exist(id)) {
4          Point deleting_point;
5          Node* deleting_node = new Node();
6          deleting_point.id = id;
7          deleting_node->point = deleting_point;
8          my_tree->eject(deleting_node);
9          delete deleting_node;
10     }
11 }
```

۴. تست کیس و مقایسه با فایل size.refout

در وهله بعد کاری که باید انجام بشه اینه که در هر استپ زمانی نقاط skyline آن محاسبه بشه (فرزندان root) و در آرایه/وکتوری از استرینگ‌ها ذخیره بشه. و در نهایت تک تک اعضای آن (که به ازای هر ایندکس نماینده هر استپ زمانی یا به عبارتی نماینده اعضای هر خط در فایل رفرنس هستند.) با فایل size.refout چک بشن. اگه عضوی باشه که با خط مربوطه در فایل رفرنس همخوانی نداشته باشه و برابر نباشه به عنوان ارور مشخص می‌شه و تعداد ارورها محاسبه شده و ارور ریت مشخص می‌شود. در نهایت هم محتوای بدست آمده در فایل size.out چاپ می‌شود.

| تحلیل نتایج اجرا

۱. الگوریتم BJR-tree

پیچیدگی زمانی:

- ایجاد درخت **BJR**: درخت BJR برای ذخیره‌سازی روابط domination بین نقاط استفاده می‌شود. هر نقطه باید به درخت اضافه شود، و در این فرایند، تمام گره‌ها باید برای بررسی روابط تسلط و تصمیم‌گیری درباره‌ی اتصال به درخت بررسی شوند.
- پیچیدگی زمانی: هر بار که یک نقطه جدید به درخت اضافه می‌شود، زمان بررسی آن به عمق درخت بستگی دارد. اگر درخت به عمق d برسد، پیچیدگی زمان اضافه کردن یک نقطه به درخت $O(d)$ خواهد بود. به دلیل اینکه درخت سلسله‌مراتبی است و روابط تسلط از ریشه به گره‌ها گسترش می‌یابد، در بهترین حالت این زمان خطی است $O(n)$ ، اما در بدترین حالت، پیچیدگی به $O(n^2)$ می‌رسد که به علت بررسی روابط تسلط برای هر نقطه است.

بهینه‌سازی:

- Lazy Injection و ND-Cache می‌توانند به کاهش زمان بررسی و بهینه‌سازی درخت کمک کنند، اما عمق درخت همچنان به‌عنوان یک عامل محدودکننده عمل می‌کند.

نتایج واقعی:

- حالت عادی برای سایزهای مختلف:

```
error rate: 0.00%
program finished
Execution time: 15 ms
Data printed in .out
Done for tree size: small, lazy: 0, nd: 0
```

```
error rate: 0.00%
program finished
Execution time: 42369 ms
Data printed in .out
Done for tree size: medium, lazy: 0, nd: 0
```

```
error rate: 0.00%
program finished
Execution time: 2326652 ms
Data printed in .out
Done for tree size: large, lazy: 0, nd: 0
```

۲. الگوریتم Lazy Injection و Injection

پیچیدگی زمانی:

- **Injection:** در روش معمولی inject، هر نقطه جدید به‌طور مستقیم به درخت اضافه می‌شود و روابط تسلط بررسی می‌شوند. این فرآیند پیچیدگی $O(d)$ دارد، که در آن d عمق درخت است. اگر درخت به‌طور کامل بررسی شود، این زمان می‌تواند به $O(n)$ برسد.
- **Lazy Injection:** در Lazy Injection، نقطه‌ها تنها در صورت نیاز به درخت اضافه می‌شوند. این باعث می‌شود که تعداد عملیات کمتر شود، زیرا تنها زمانی که یک نقطه احتمال وجود در Skyline را داشته باشد، به درخت اضافه می‌شود. این استراتژی باعث می‌شود که پیچیدگی زمان در عمل کاهش یابد.
 - با این حال، در بدترین حالت همچنان پیچیدگی زمان می‌تواند به $O(n)$ برسد زیرا ممکن است برای بررسی کامل تمام نقاط نیاز به مراجعه به کل درخت باشد.

بهینه‌سازی:

Data Structures Project Report

- استفاده از Lazy Injection باعث کاهش پیچیدگی عملیاتی می‌شود، زیرا عملیات‌های اضافی حذف می‌شوند و فقط زمانی که به‌طور واقعی لازم است، نقاط وارد درخت می‌شوند.

نتایج واقعی:

- نتایج برای سایزهای مختلف dataset:

```
error rate: 0.00%
program finished
Execution time: 12 ms
Data printed in .out
Done for tree size: small, lazy: 1, nd: 0
```

```
error rate: 0.00%
program finished
Execution time: 44390 ms
Data printed in .out
Done for tree size: medium, lazy: 1, nd: 0
```

```
error rate: 0.00%
program finished
Execution time: 2550227 ms
Data printed in .out
Done for tree size: large, lazy: 1, nd: 0
```

۳. ND-Cache

پیچیدگی زمانی:

- ND-Cache برای ذخیره‌سازی اطلاعات non-dominance استفاده می‌شود تا از محاسبات دوباره جلوگیری کند. این کش به‌ویژه در مراحل پیوسته (زمانی که نقاط به‌طور مداوم وارد و خارج می‌شوند) مفید است.

Data Structures Project Report

- پیچیدگی زمانی: دسترسی به کش بسیار سریع است و پیچیدگی زمانی آن $O(1)$ خواهد بود. بنابراین، از نظر زمانی، استفاده از کش می‌تواند تأثیر زیادی در کاهش زمان محاسباتی داشته باشد.
- حجم کش: کش به اندازه‌ای بزرگ است که باید با توجه به تعداد نقاط موجود تنظیم شود. اگر تعداد نقاط زیاد باشد، کش ممکن است نیاز به به‌روزرسانی مداوم داشته باشد.

بهینه‌سازی:

- با استفاده از کش، تعداد مقایسه‌ها به‌طور چشمگیری کاهش می‌یابد، زیرا اطلاعات مربوط به روابط non-dominance قبلاً ذخیره شده است و نیازی به محاسبه مجدد نیست.

نتایج واقعی:

- نتایج برای سایزهای مختلف وقتی فقط nd فعال است:

```
error rate: 0.00%
program finished
Execution time: 13 ms
Data printed in .out
Done for tree size: small, lazy: 0, nd: 1
```

```
error rate: 0.00%
program finished
Execution time: 43024 ms
Data printed in .out
Done for tree size: medium, lazy: 0, nd: 1
```

```
Error rate: 0%
program finished
Execution time: 2349265 ms
Data printed in .out
Done for tree size: large, lazy: 0, nd: 1
```

- و برای زمانی که هم از nd و هم از lazy evaluation استفاده می‌کنیم:

```
error rate: 0.00%
program finished
Execution time: 12 ms
Data printed in .out
Done for tree size: small, lazy: 1, nd: 1
```

```
error rate: 0.00%
program finished
Execution time: 45505 ms
Data printed in .out
Done for tree size: medium, lazy: 1, nd: 1
```

```
Error rate: 0%
program finished
Execution time: 2618344 ms
Data printed in .out
Done for tree size: large, lazy: 1, nd: 1
```

۵. عملکرد کلی

با توجه به تمام پیچیدگی‌های الگوریتم‌ها، درخت BJR و الگوریتم‌های مرتبط با آن، پیچیدگی کلی فرآیند به شدت تحت تأثیر:

- **عمق درخت BJR**: هرچه عمق درخت بیشتر باشد، تعداد عملیات‌های لازم برای بررسی روابط تسلط و انجام عملیات‌ها بیشتر خواهد بود.
- **Lazy Injection**: به دلیل انجام تنها عملیات‌های ضروری، باعث کاهش قابل توجه در تعداد عملیات‌ها می‌شود.
- **ND-Cache**: سرعت قابل توجهی در دسترسی به اطلاعات قبلی و جلوگیری از محاسبات مجدد ایجاد می‌کند.
- **Quick Sort**: برای مرتب‌سازی نقاط، در حالت عمومی پیچیدگی $O(n \log n)$ خواهد بود.