Cấu trúc Trie

26/12/2024

23120143 – Tô Thành Long
23120143@student.hcmus.edu.vn

fit@hcmus

Đồ án điểm cộng môn học

Trie trong autocomplete

Môn Thực hành Cấu trúc dữ liệu & Giải thuật

fit@hcmus

Nội dung trình bày

01 Mã nguồn đồ án

- Mô tả cấu trúc Trie
- Mô tả cấu trúc Sorted Array
- Mô tả cấu trúc Cache Manager
- Mô tả thuật toán Fuzzy Search
- Mô tả thuật toán Search By Regex
- Mô tả giao diện
- Dữ liệu và thư viện được sử dụng

02 Phân tích và thực nghiệm

- Phân tích thuật toán trên lý thuyết
- Đánh giá thực toán trên thực nghiệm

03 Tài liệu tham khảo

- Tài liệu văn bản
- Tài liệu ảnh

01. Mã nguồn đồ án

01. Mã nguồn đồ án

Mô tả cấu trúc Trie

Mô tả cấu trúc Trie

Giới thiệu sơ lược

- **Trie**[1] (còn được gọi là prefix tree hoặc digital tree) là một cấu trúc dữ liệu dạng cây, được sử dụng để lưu trữ và truy xuất các chuỗi một cách hiệu quả, đặc biệt là khi làm việc với các chuỗi có chung tiền tố.
- Về cấu trúc, Trie bao gồm:
  - Node gốc (root)
  - Các nút con (children)
  - Đánh dấu kết thúc chuỗi (end-of-word marker)
- Các thao tác cơ bản trên trie:
  - Thêm một từ
  - Xoá một từ
  - Tìm kiếm một từ
  - Tìm kiếm tiền tố

Mô tả cấu trúc Trie

Tổ chức mã nguồn

struct TrieNode

- TrieNode* children[26]
- bool isEndOfWorld

class Trie

- TrieNode* root
- CacheManager* cache
- void loadDictionary()
- void insert()
- void remove()
- vector<string> suggest()
- vector<string> fuzzySearch()

** Mô tả đầy đủ ở báo cáo, p.1*

Mô tả cấu trúc Trie

Tổ chức mã nguồn — **Thực nghiệm**

class TrieUnitTests

- void testInsertion()
- void testSearchWithExistingWord()
- void testSearchWithNonExistingWord()
- void testRemoval()
- void testEmptiness()
- void testSuggestNoRegex()
- void testSuggestWithRegex()
- void testFuzzySearch()

Mô tả cấu trúc Trie

Tổ chức mã nguồn — **Thực nghiệm**

class TriePerformanceTests

- void runAllTest()
- void testInsertion()
- void testSuggest()
- void testRemoval()

01. Mã nguồn đồ án

Mô tả cấu trúc Sorted Array

Mô tả cấu trúc Sorted Array

Giới thiệu sơ lược

- **Sorted Array** (còn được gọi là mảng đã được sắp xếp) là một mảng chứa các chuỗi ký tự được sắp xếp theo thứ tự từ điển (lexicographical order). Cấu trúc này sẽ được sử dụng trong đồ án để so sánh với Trie thông qua các chức năng cơ bản là **thêm từ, xoá từ và đề xuất từ.**

Mô tả cấu trúc Sorted Array

Giới thiệu sơ lược

- Quá trình tìm kiếm các từ bắt đầu bằng tiền tố cho trước đồng nghĩa với việc một khoảng (mảng con) mà các từ thuộc khoảng này phải bắt đầu bằng tiền tố. Việc tìm kiếm này có thể thực hiện bằng cách sử dụng **Binary Search** để tìm chặn trên và chặn dưới của khoảng, sau đó liệt kê các từ thuộc khoảng này.

Các từ bắt đầu bằng "ab"

Mô tả cấu trúc Sorted Array

Tổ chức mã nguồn

class SortedArray

- vector<string> words
- void loadDictionary()
- void insert()
- void remove()
- vector<string> suggest()

** Mô tả đầy đủ trong báo cáo, p.3*

Mô tả cấu trúc Sorted Array

Tổ chức mã nguồn — **Thực nghiệm**

class SortedArrayPerformanceTests

- void runAllTest()
- void testInsertion()
- void testSuggest()
- void testRemoval()

01. Mã nguồn đồ án

Mô tả cấu trúc Cache Manager

Mô tả cấu trúc Cache Manager

Giới thiệu sơ lược

- Đây là cấu trúc được thiết kế để lưu trữ và quản lý các từ được đề xuất liên quan đến tiền tố mà người dùng đã tìm kiếm. Chức năng chính của nó là tối ưu hóa hiệu suất trong việc cung cấp các đề xuất, bằng cách duy trì một bộ nhớ đệm với các quy tắc ưu tiên dựa trên tần suất tìm kiếm.
- **Cache Manager** sử dụng một **Hash Map** lưu trữ cặp ***(prefix, {suggestedWord, frequency})*** để theo dõi tần suất tìm kiếm của tiền tố và một biến capacity để giới hạn dung lượng trong cache.

Mô tả cấu trúc Cache Manager

Giới thiệu sơ lược

ab	• words = {ability, absence, absolute, abstract} • freq = 5
ac	• words = {accept, academy, access} • freq = 4
pre	• words = {preach, precede, precise, predict, prefer, prepare} • freq = 3
comp	• words = {compact, company, compare, compel, compile} • freq = 2
sta	• words = {stable, stack, staff, stain, stake} • freq = 1
...	• ...

Mô tả cấu trúc Cache Manager

Giới thiệu sơ lược

- **Các chức năng chính:**
  - Thêm tiền tố và từ được đề xuất vào cache:
    - Nếu tiền tố đã tồn tại trong cache, tăng tần suất tìm kiếm (frequency).
    - Nếu tiền tố chưa tồn tại, thêm tiền tố và các từ được đề xuất vào Hash Map. Trong trường hợp cache vượt quá dung lượng giới hạn, loại bỏ tiền tố có tần suất thấp nhất.
  - Lấy danh sách từ được đề xuất:
    - Nếu tiền tố đã tồn tại trong cache, trả về danh sách các từ đã được lưu.
    - Nếu không tồn tại, thực hiện truy vấn trong Trie để lấy danh sách đề xuất, sau đó thêm vào cache.

Mô tả cấu trúc Cache Manager

Giới thiệu sơ lược

- **Các chức năng chính:**
  - Làm mới tiền tố:
    - Trong trường hợp một từ được thêm hoặc xoá trên Trie, điều này sẽ ảnh hưởng đến các từ đề xuất dựa trên tiền tố đã được lưu trong cache. Do đó, khi thêm hoặc xoá một từ, CacheManager sẽ tiến hành xoá các tiền tố có chứa từ này, sau đó tiến hành cập nhật lại khi người dùng tìm kiếm.

Bằng việc tích hợp Cache Manager vào trong Trie, các tiền tố có tần suất tìm kiếm cao sẽ được lưu giữ lâu hơn trong cache, đảm bảo rằng các đề xuất phổ biến luôn sẵn sàng mà không cần truy vấn lại từ Trie. Điều này cải thiện hiệu suất cho các trường hợp người dùng liên tục tìm kiếm những tiền tố phổ biến.

Mô tả cấu trúc Cache Manager

Tổ chức mã nguồn

struct CacheNode

- vector<string> suggestions
- int frequency

class CacheManager

- unordered_map<string, CacheNode> cache
- int size
- int capacity
- void insert()
- void update()
- vector<string> get()

** Mô tả đầy đủ trong báo cáo, p.4*

Mô tả cấu trúc Cache Manager

Tổ chức mã nguồn

struct CacheNode

- vector<string> suggestions
- int frequency

class CacheManager

- void remove()
- void removeItemByWord()
- void evict()

** Mô tả đầy đủ trong báo cáo, p.4*

01. Mã nguồn đồ án

Mô tả thuật toán Fuzzy Search

Mô tả thuật toán Fuzzy Search

Giới thiệu sơ lược

- **Thuật toán tìm kiếm mờ (Fuzzy Search)**[3] được sử dụng để tìm kiếm các từ gần khớp trong cấu trúc dữ liệu Trie mà có thể sai khác với từ khóa tìm kiếm một khoảng cách chỉnh sửa nhỏ nhất định. Thuật toán này áp dụng khoảng cách Levenshtein để tính số bước tối thiểu (thêm, xóa, hoặc thay thế ký tự) cần thiết để biến một chuỗi thành chuỗi khác.
- *Ví dụ: Với từ "apple", các từ khoá gần khớp mà người dùng có thể nhập là "apl", "appke", "apples",...*

Mô tả thuật toán Fuzzy Search

Giới thiệu sơ lược

- **Khoảng cách Levenshtein**[4] là một thước đo về độ tương đồng giữa hai chuỗi. Nó được định nghĩa là số lượng tối thiểu các phép chỉnh sửa ký tự cần thiết để biến một chuỗi thành một chuỗi khác. Các phép chỉnh sửa bao gồm:
  - Thêm một ký tự vào chuỗi (insert).
  - Xóa một ký tự khỏi chuỗi (delete).
  - Thay một ký tự này bằng một ký tự khác (replace).

Mô tả thuật toán Fuzzy Search

Giới thiệu sơ lược

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Hình 1: Công thức tính khoảng các Levenshtein giữa 2 chuỗi a và b[6]

Mô tả thuật toán Fuzzy Search

Giới thiệu sơ lược

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	**3**

Ô dưới cùng góc phải chính là khoảng cách Levenshtein giữa 2 từ này, hoặc nói các khác là số bước chỉnh sửa tối thiểu để biến "kitten" thành "sitting".

Mô tả thuật toán Fuzzy Search

Cách thức hoạt động

- Tính khoảng cách Levenshtein:
  - Dùng mảng quy hoạch động để tính khoảng cách Levenshtein giữa tiền tố của từ trong Trie (currentWord) và tiền tố của từ khoá tìm kiếm (query).
  - Mỗi ô trong mảng thể hiện số bước cần thiết để chuyển đổi tiền tố của currentWord thành tiền tố của query.
- Kiểm tra kết quả:
  - Nếu khoảng cách Levenshtein tại ô cuối cùng của dòng hiện tại trong mảng 2 chiều nhỏ hơn hoặc bằng maxDistance và kí tự hiện tại là kết thúc của một từ, thì từ này được lưu lại.

Mô tả thuật toán Fuzzy Search

Cách thức hoạt động

- Loại bỏ trường hợp:
  - Nếu ô có khoảng cách Levenshtein nhỏ nhất có giá trị lớn hơn maxDistance thì kể từ kí tự này chắc chắn không có kết quả, nên ta sẽ loại bỏ nhánh này.
- Tiếp tục kiểm tra các kí tự tiếp theo:
  - Với nỗi nút (đại diện cho 1 kí tự), ta tiếp tục đệ quy xuống các nút con để tạo thành currentWord mới và tiếp tục tiến hành kiểm tra.

Vì vậy, tìm kiếm mờ trong Trie rất hiệu quả vì chỉ cần duyệt các nhánh có khả năng chứa kết quả và cung cấp một thước đo chính xác về độ tương đồng giữa các chuỗi.

01. Mã nguồn đồ án

Mô tả thuật toán Search By Regex

Mô tả thuật toán Search By Regex

Giới thiệu sơ lược

- **Biểu thức chính quy (Regular Expression)**[5], thường được gọi tắt là regex, là một chuỗi các ký tự đặc biệt định nghĩa một mẫu tìm kiếm. Nó là một công cụ cực kỳ mạnh mẽ để làm việc với chuỗi, cho phép:
  - Tìm kiếm sự xuất hiện của một mẫu cụ thể trong một chuỗi lớn.
  - Thay thế các phần của chuỗi khớp với một mẫu bằng một chuỗi khác.
  - Kiểm tra xem một chuỗi có tuân theo một định dạng nhất định hay không.
  - Trích xuất các phần cụ thể của chuỗi khớp với một mẫu.

Mô tả thuật toán Search By Regex

Giới thiệu sơ lược

- **Regex** được sử dụng rộng rãi trong nhiều ngôn ngữ lập trình, trình soạn thảo văn bản và công cụ dòng lệnh. Regex hoạt động bằng cách sử dụng các ký tự đại diện và các ký tự thông thường để tạo thành một mẫu. Một công cụ sẽ so sánh mẫu này với chuỗi đầu vào để tìm ra các vị trí khớp. Trong đồ án này, hàm searchByRegex() chỉ sử dụng kí tự đại diện **"." (dấu chấm), "^" (dấu mũ)** và **"[]" (dấu ngoặc vuông).** Lưu ý, kí tự dấu mũ sẽ luôn đi cùng với dấu ngoặc vuông trong đồ án này.
- *Ví dụ với regex "a..le", các từ khớp là "apple", "arole"; với regex "b[aeiou]g", các từ khớp là "bag", "beg", "big", "bog", "bug".*

Mô tả thuật toán Search By Regex

Cách thức hoạt động

- Ba kí tự đại diện được sử dụng sẽ thực hiện các chức năng tương ứng như sau:
  - Đối với kí tự dấu chấm, nó sẽ so khớp với tất cả các kí tự, tức là hàm sẽ thử hết 26 kí tự trong mảng children.
  - Đối với kí tự dấu ngoặc vuông, nó sẽ so khớp với từng kí tự bên trong dấu ngoặc vuông.
  - Đối với kí tự dấu mũ, nó sẽ loại trừ các kí tự bên trong ngoặc vuông, tức là so khớp với các kí tự còn lại trong 26 kí tự.
  - Đối với kí tự khác, nó sẽ duyệt như bình thường.

Regex *là một công cụ mạnh mẽ để làm việc với chuỗi, và ký tự dấu chấm,dấu ngoặc vuông và dấu mũ là những thành phần cơ bản của nó. Trong Trie, việc sử dụng 3 kí tự cho phép người dùng thực hiện tìm kiếm một cách linh hoạt và hiệu quả hơn.*

01. Mã nguồn đồ án

Mô tả giao diện

Mô tả giao diện

Tổ chức mã nguồn

class UI

- void userMode()
- void trieTestMode()
- void sortedArrayTestMode()
- void run()

01. Mã nguồn đồ án

Dữ liệu và thư viện được sử dụng

Dữ liệu và thư viện được sử dụng

Đồ án sử dụng file **words_alpha.txt** với 370104 từ (danh sách các từ tiếng Anh chỉ gồm các chữ Latin) được lấy từ repository https://github.com/dwyl/english-words

Các cấu trúc

- <iostream>
- <fstream>
- <vector>
- <algorithm>
- <unordered_map>
- <climits>
- <string>

Thực nghiệm & đo đạc

- <cassert>
- <chrono>

Giao diện

- <conio.h>

02. Phân tích và thực nghiệm

fit@hcmus

02. Phân tích và thực nghiệm

Phân tích thuật toán trên lý thuyết

Phân tích thuật toán trên lý thuyết

Độ phức tạp và phép so sánh của hàm *insert()* với từ có *n* kí tự

	Trie	Sorted Array
Độ phức tạp	O(n)	O(H)
Số phép so sánh	n	2 * logH + H + 2

** H là kích thước mảng (tức số lượng từ trong từ điển). Phần chứng minh từ p.9 – p.10*

Phân tích thuật toán trên lý thuyết

Độ phức tạp và phép so sánh của hàm *insert()* với từ có *n* kí tự

n	Trie	Sorted Array
0	0	21
25	25	370143
50	50	370143
75	75	370143
100	100	370143
125	125	370143
150	150	370143
175	175	370143
200	200	370143
225	225	370143
250	250	370143

n	Trie	Sorted Array
275	275	370143
300	300	370143
325	325	370143
350	350	370143
375	375	370143
400	400	370143
425	425	370143
450	450	370143
475	475	370143
500	500	370143

Phân tích thuật toán trên lý thuyết

Độ phức tạp và phép so sánh của hàm *insert()* với từ có *n* kí tự

Phân tích thuật toán trên lý thuyết

Độ phức tạp và phép so sánh của hàm *remove()* với từ có *n* kí tự

	Trie	Sorted Array
Độ phức tạp	O(n)	O(H)
Số phép so sánh	28 * n - 27	2 * logH + H + 2

** H là kích thước mảng (tức số lượng từ trong từ điển). Phần chứng minh từ p.11 – p.12*

Phân tích thuật toán

Độ phức tạp và phép so sánh của hàm *remove()* với từ có *n* kí tự

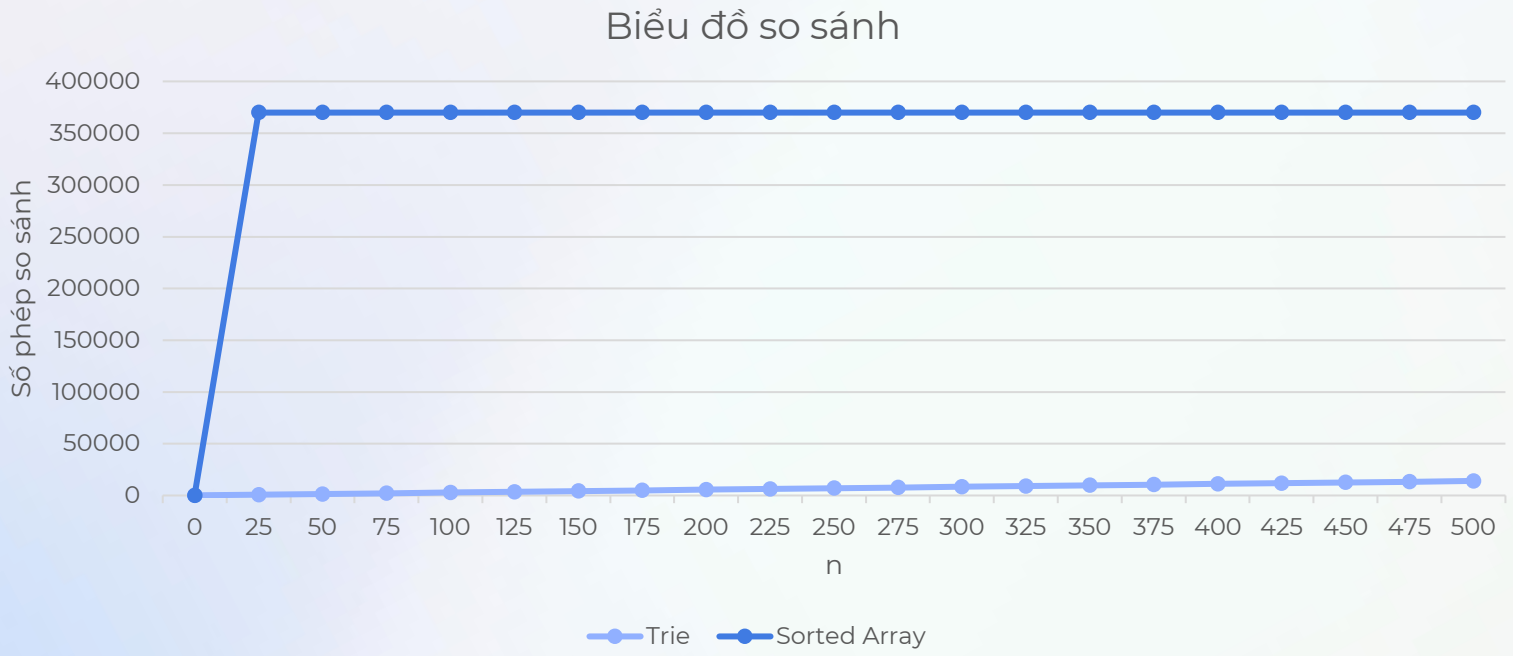
n	Trie	Sorted Array
0	0	21
25	673	370143
50	1373	370143
75	2073	370143
100	2773	370143
125	3473	370143
150	4173	370143
175	4873	370143
200	5573	370143
225	6273	370143
250	6973	370143

n	Trie	Sorted Array
275	7673	370143
300	8373	370143
325	9073	370143
350	9773	370143
375	10473	370143
400	11173	370143
425	11873	370143
450	12573	370143
475	13273	370143
500	13973	370143

fit@hcmus

Phân tích thuật toán trên lý thuyết

Độ phức tạp và phép so sánh của hàm *remove()* với từ có *n* kí tự



Phân tích thuật toán trên lý thuyết

Độ phức tạp và phép so sánh của hàm *suggest()* đề xuất *wordLimit* từ, với prefix có *m* kí tự.

	Trie	Sorted Array
Độ phức tạp	O(m + 26 * L * wordLimit)	O(m * logH + wordLimit)
Số phép so sánh	m + 55 * L * wordLimit	(2 * m + 5) * logH + 4 + (wordLimit + 1) * 2

** H là kích thước mảng (tức số lượng từ trong từ điển). Phần chứng minh từ p.13 – p.17*

Phân tích thuật toán trên lý thuyết

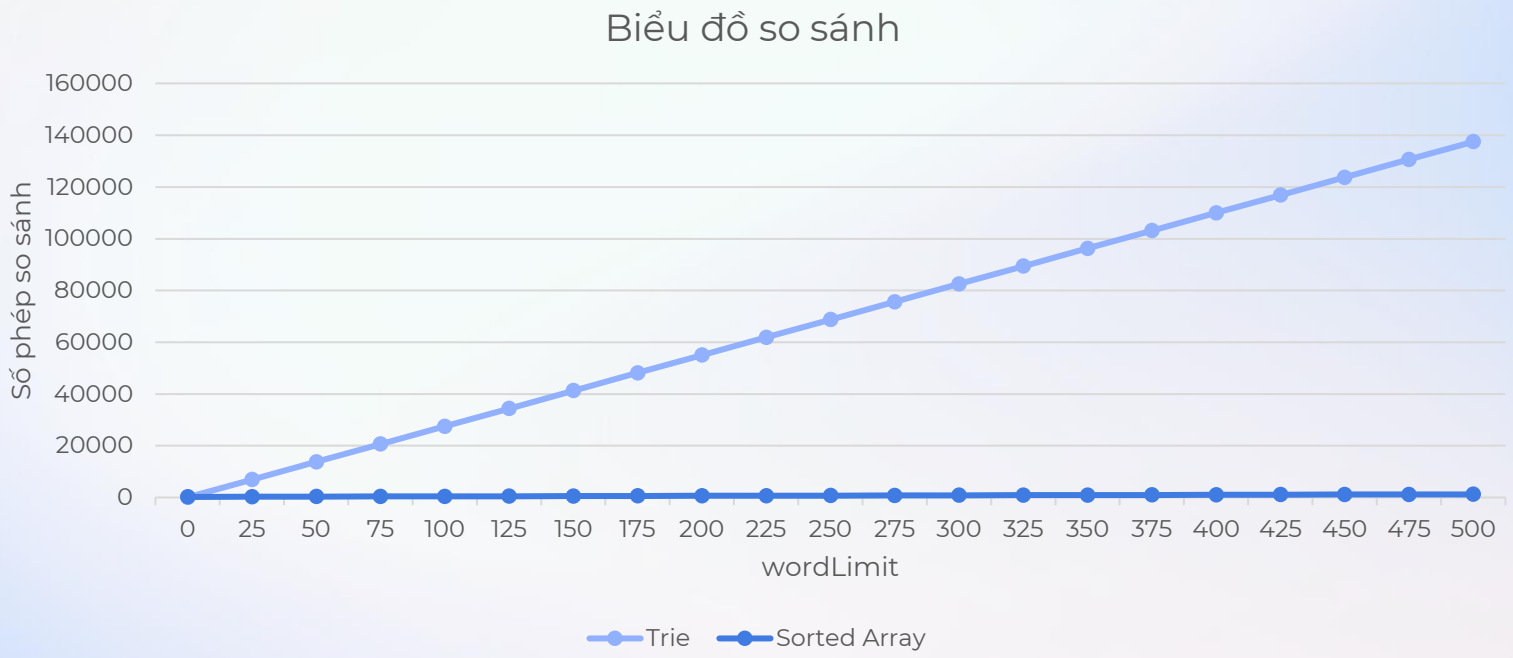
Độ phức tạp và phép so sánh của hàm *suggest()* đề xuất *wordLimit* từ, với prefix có *m* kí tự.

n	Trie	Sorted Array
0	6	215
25	6881	265
50	13756	315
75	20631	365
100	27506	415
125	34381	465
150	41256	515
175	48131	565
200	55006	615
225	61881	665
250	68756	715

n	Trie	Sorted Array
275	75631	765
300	82506	815
325	89381	865
350	96256	915
375	103131	965
400	110006	1015
425	116881	1065
450	123756	1115
475	130631	1165
500	137506	1215

Phân tích thuật toán trên lý thuyết

Độ phức tạp và phép so sánh của hàm *suggest()* đề xuất *wordLimit* từ, với prefix có *m* kí tự.



02. Phân tích và thực nghiệm

Đánh giá thuật toán trên thực nghiệm

Đánh giá thuật toán trên thực nghiệm

Đánh giá tổng quát — **Độ phức tạp**

	Trie	Sorted Array
Chèn (Insert)	O(n)	O(H)
Xoá (Remove)	O(n)	O(H)
Đề xuất (Suggest)	O(m + 26 * L * wordLimit)	O(m * logH + wordLimit)

** **n** là số kí tự của một từ; **H** là kích thước từ mảng; **m** là chiều dài prefix; **L** là số kí tự trung bình của một từ; **wordLimit** là số lượng từ giới hạn khi đề xuất*

Đánh giá thuật toán trên thực nghiệm

Đánh giá tổng quát — **Phép so sánh**

	Trie	Sorted Array
Chèn (Insert)	n	2 * logH + H + 2
Xoá (Remove)	28 * n - 27	2 * logH + H + 2
Đề xuất (Suggest)	2m + 55 * L * wordLimit	(2m + 5) * logH + 4 + (wordLimit + 1) * 2

** **n** là số kí tự của một từ; **H** là kích thước từ mảng; **m** là chiều dài prefix; **L** là số kí tự trung bình của một từ; **wordLimit** là số lượng từ giới hạn khi đề xuất*

Đánh giá thuật toán trên thực nghiệm

Đánh giá tổng quát

Nhận xét

- Thao tác chèn của Trie **nhanh hơn rất nhiều** so với Sorted Array vì giá trị H là rất lớn.
- Thao tác xoá của Trie **nhanh hơn rất nhiều** so với Sorted Array vì giá trị H là rất lớn.
- Thao tác đề xuất của Trie **chậm hơn rất nhiều** so với Sorted Array khi wordLimit càng lớn.

Đánh giá thuật toán

Số liệu thực nghiệm của số phép so sánh

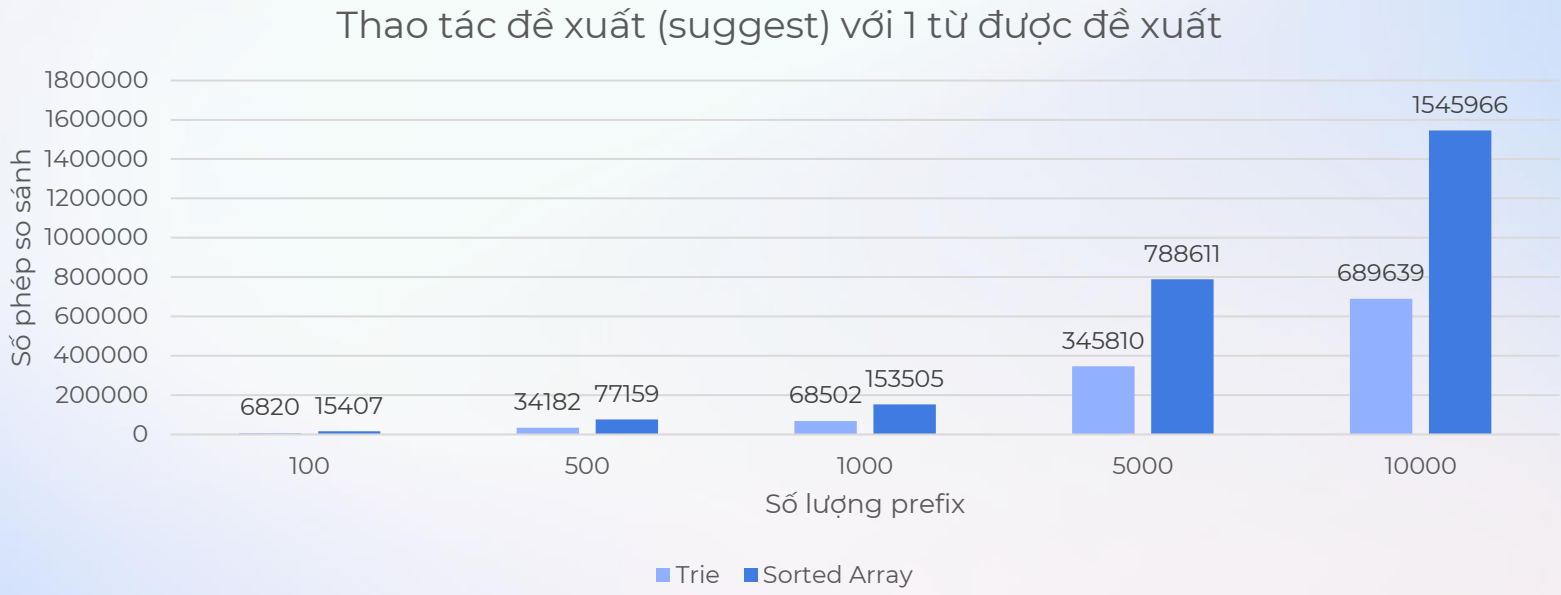
Nhận xét

- Ta thực hiện chạy hàm với số lượng từ/prefix lần lượt là **100, 500, 1000, 5000, 10000**. Đối với hàm suggest(), ta tiến hành đề xuất số từ lần lượt là **1, 5, 10, 15, 20.**

Đánh giá thuật toán

Số liệu thực nghiệm của số phép so sánh

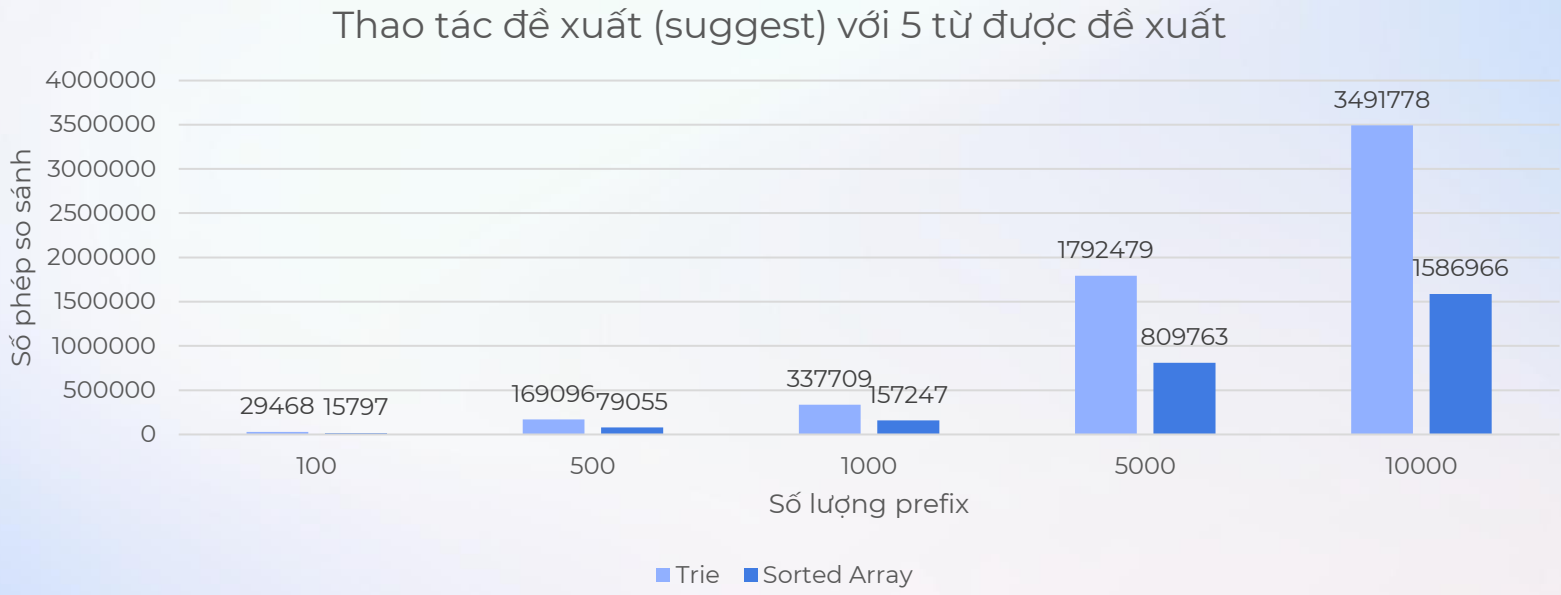
Với thao tác đề xuất (suggest), ở đây các prefix được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh số phép so sánh giữa Trie với Sorted Array như sau:



Đánh giá thuật toán

Số liệu thực nghiệm của số phép so sánh

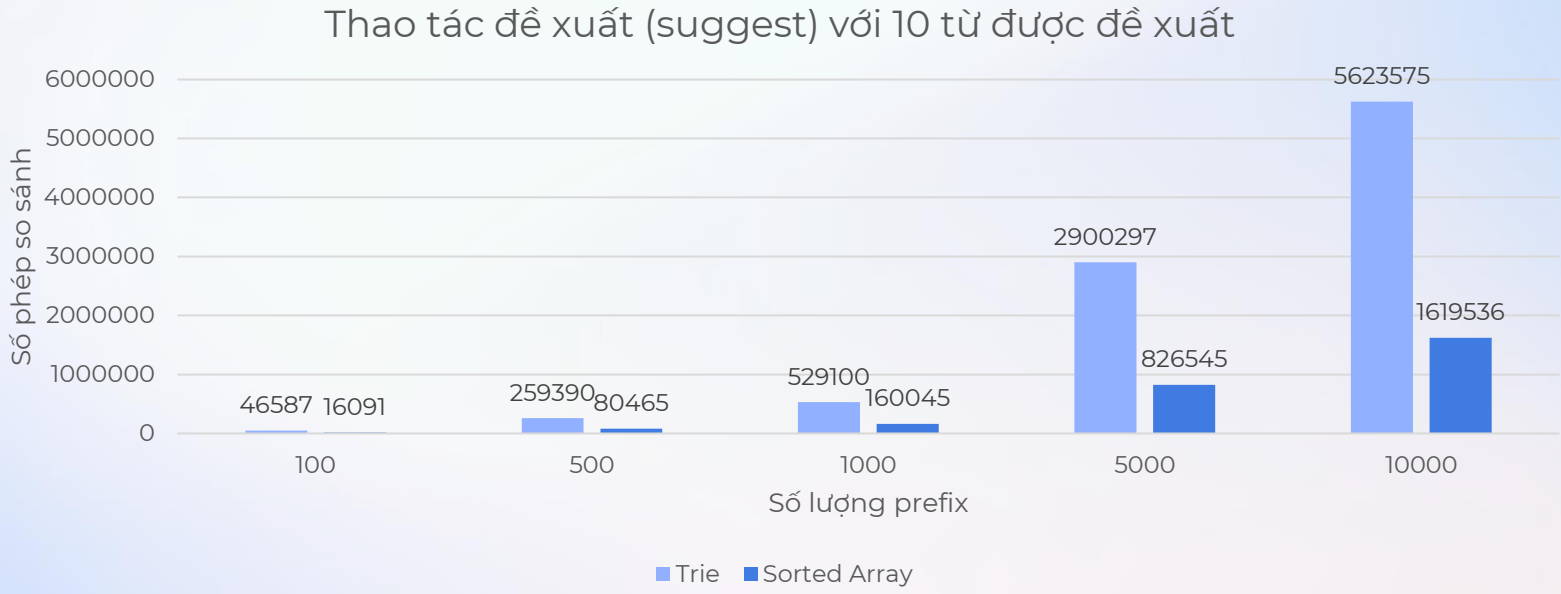
Với thao tác đề xuất (suggest), ở đây các prefix được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh số phép so sánh giữa Trie với Sorted Array như sau:



Đánh giá thuật toán

Số liệu thực nghiệm của số phép so sánh

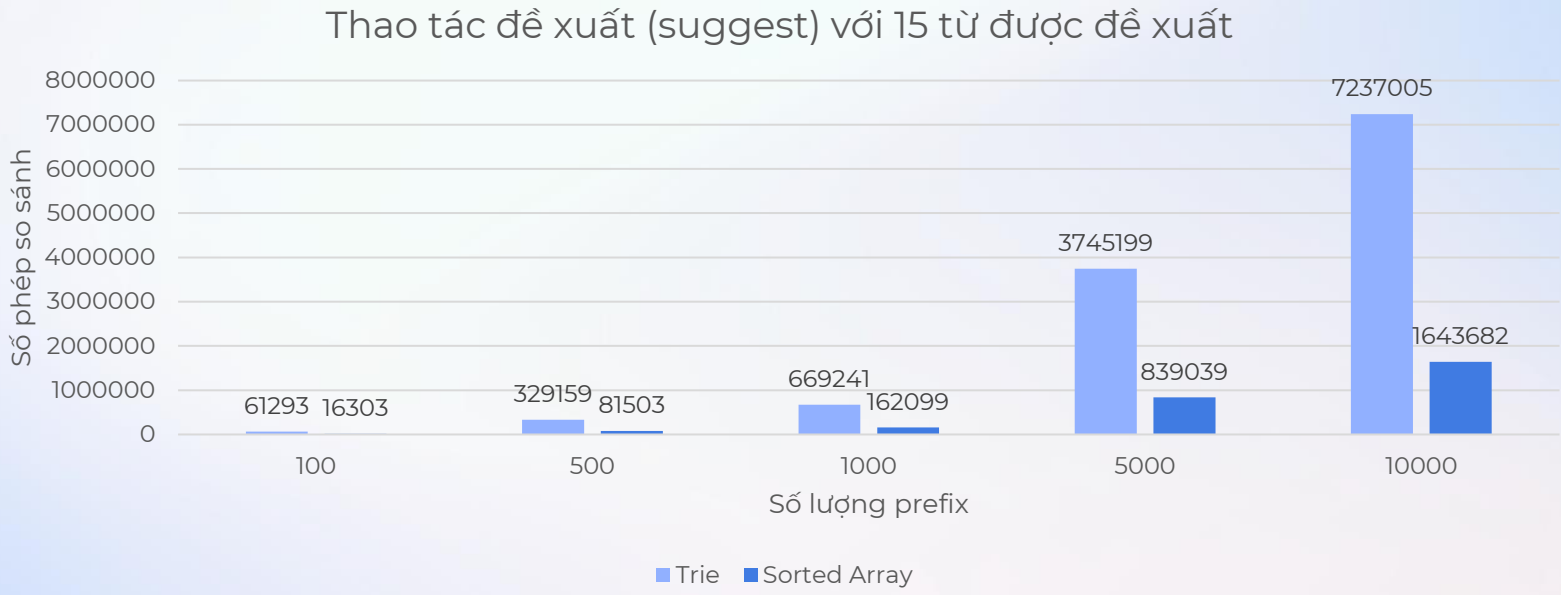
Với thao tác đề xuất (suggest), ở đây các prefix được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh số phép so sánh giữa Trie với Sorted Array như sau:



Đánh giá thuật toán

Số liệu thực nghiệm của số phép so sánh

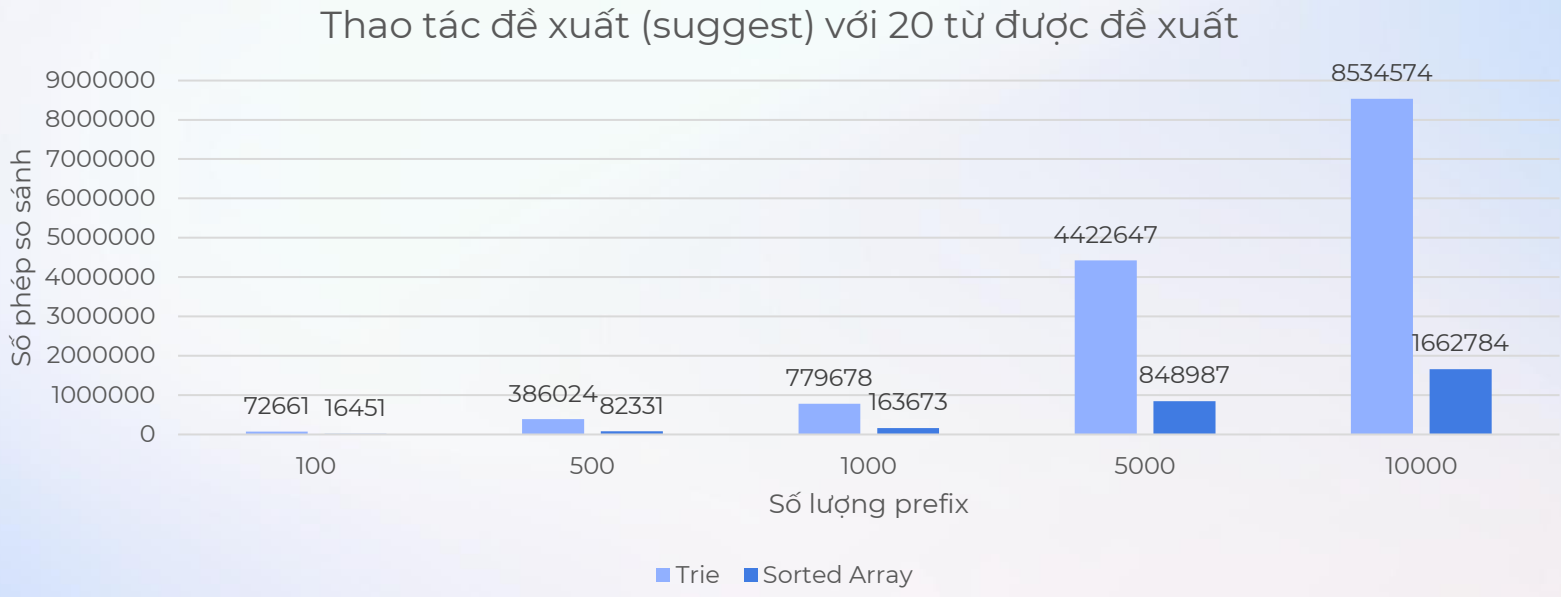
Với thao tác đề xuất (suggest), ở đây các prefix được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh số phép so sánh giữa Trie với Sorted Array như sau:



Đánh giá thuật toán

Số liệu thực nghiệm của số phép so sánh

Với thao tác đề xuất (suggest), ở đây các prefix được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh số phép so sánh giữa Trie với Sorted Array như sau:



Đánh giá thuật toán

Số liệu thực nghiệm của thời gian thực thi

Nhận xét

- Để tính chính xác hơn thời gian thực thi của các hàm, ta sẽ thực tiến hành chạy hàm **10 lần** với số lượng từ/prefix lần lượt là **100, 500, 1000, 5000, 10000**. Sau đó ta tính trung bình thời gian chạy 10 lần đó để lấy kết quả thực nghiệm. Đối với hàm suggest(), ta tiến hành đề xuất số từ lần lượt là **1, 5, 10, 15, 20.**

Đánh giá thuật toán

Số liệu thực nghiệm của thời gian thực thi

Với thao tác chèn (insert), biểu đồ so sánh thời gian thực hiện trung bình giữa Trie với Sorted Array như sau:

Đánh giá thuật toán

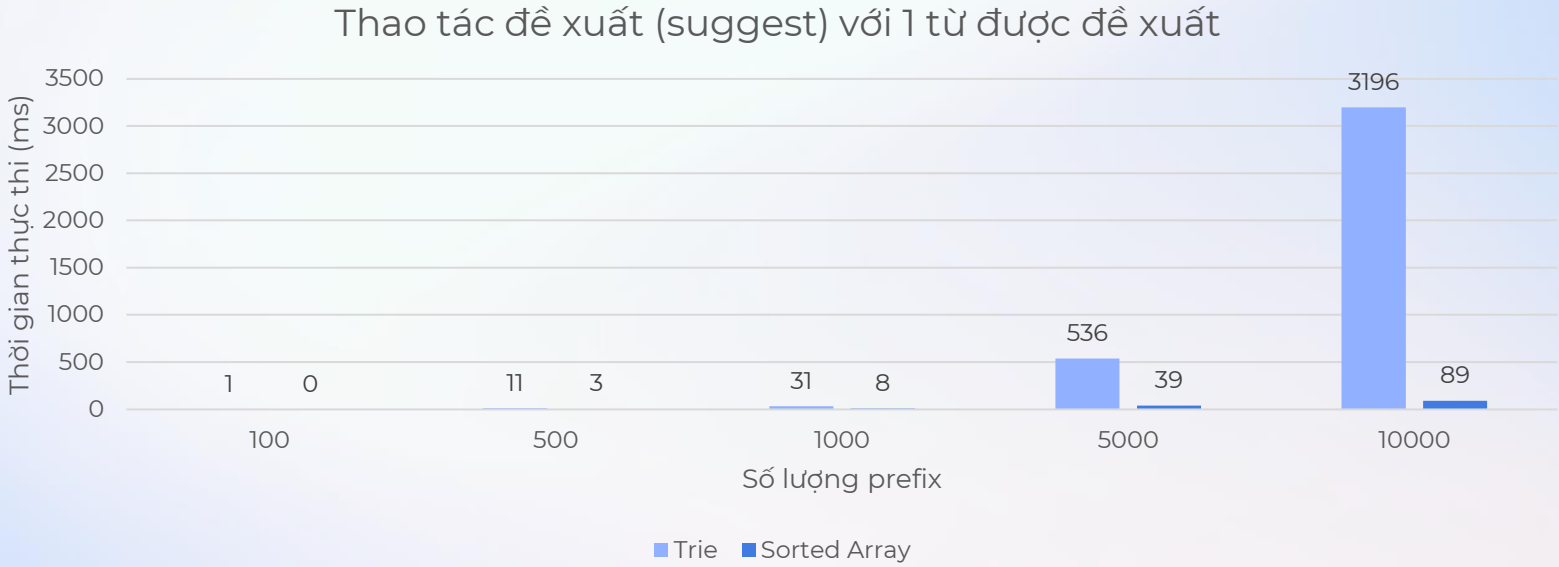
Số liệu thực nghiệm của thời gian thực thi

Với thao tác xoá (remove), biểu đồ so sánh thời gian thực hiện trung bình giữa Trie với Sorted Array như sau:

Đánh giá thuật toán

Số liệu thực nghiệm của thời gian thực thi

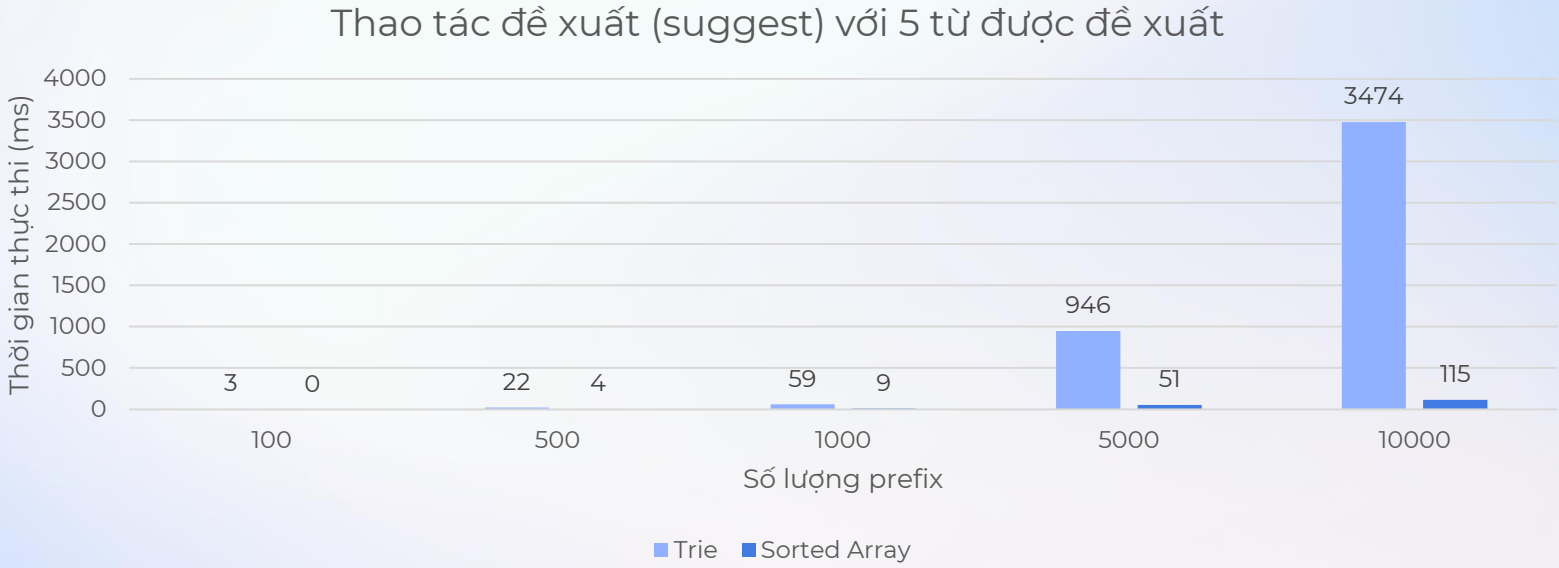
Với thao tác đề xuất (suggest), ở đây các prefix được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh thời gian thực hiện trung bình giữa Trie với Sorted Array như sau:



Đánh giá thuật toán

Số liệu thực nghiệm của thời gian thực thi

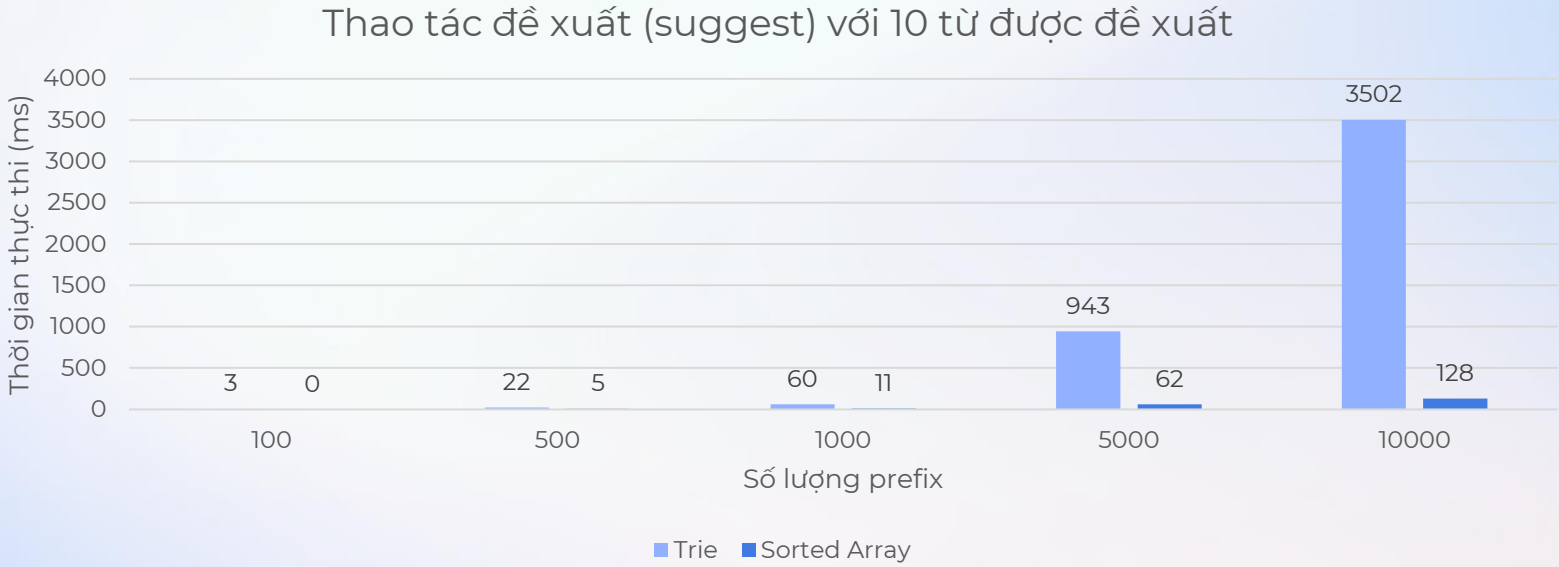
Với thao tác đề xuất (suggest), ở đây các prefix được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh thời gian thực hiện trung bình giữa Trie với Sorted Array như sau:



Đánh giá thuật toán

Số liệu thực nghiệm của thời gian thực thi

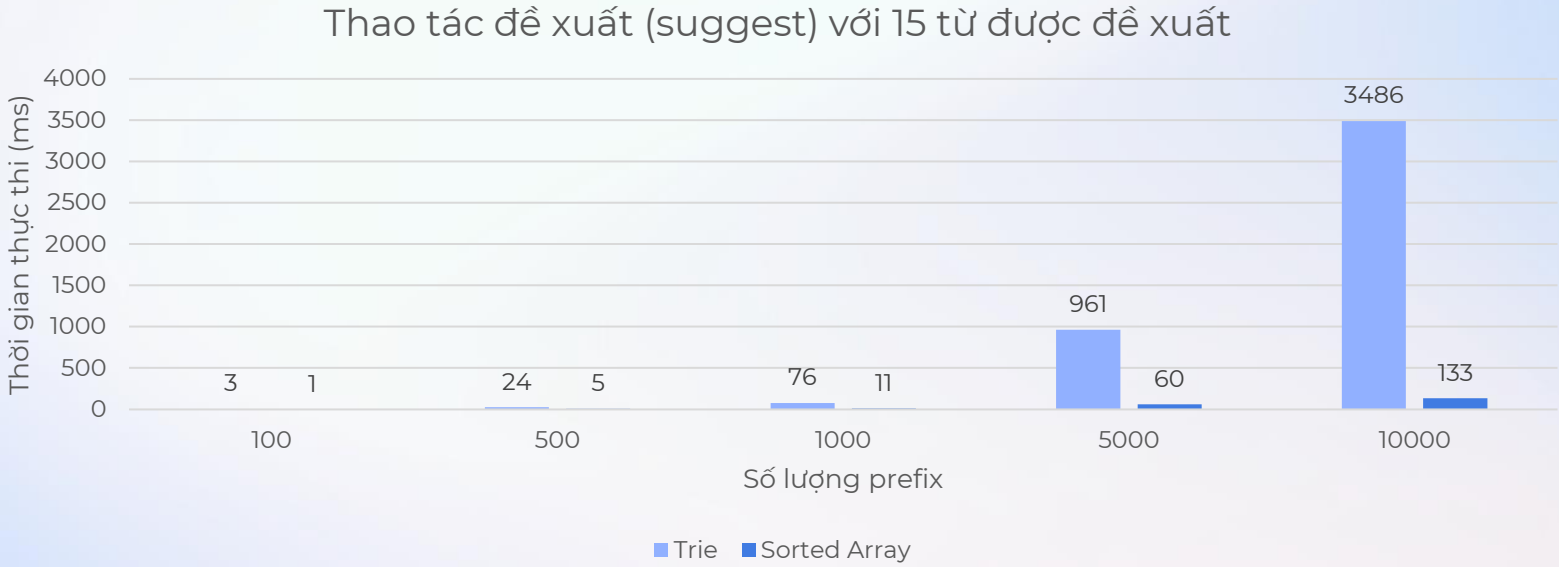
Với thao tác đề xuất (suggest), ở đây các prefix được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh thời gian thực hiện trung bình giữa Trie với Sorted Array như sau:



Đánh giá thuật toán

Số liệu thực nghiệm của thời gian thực thi

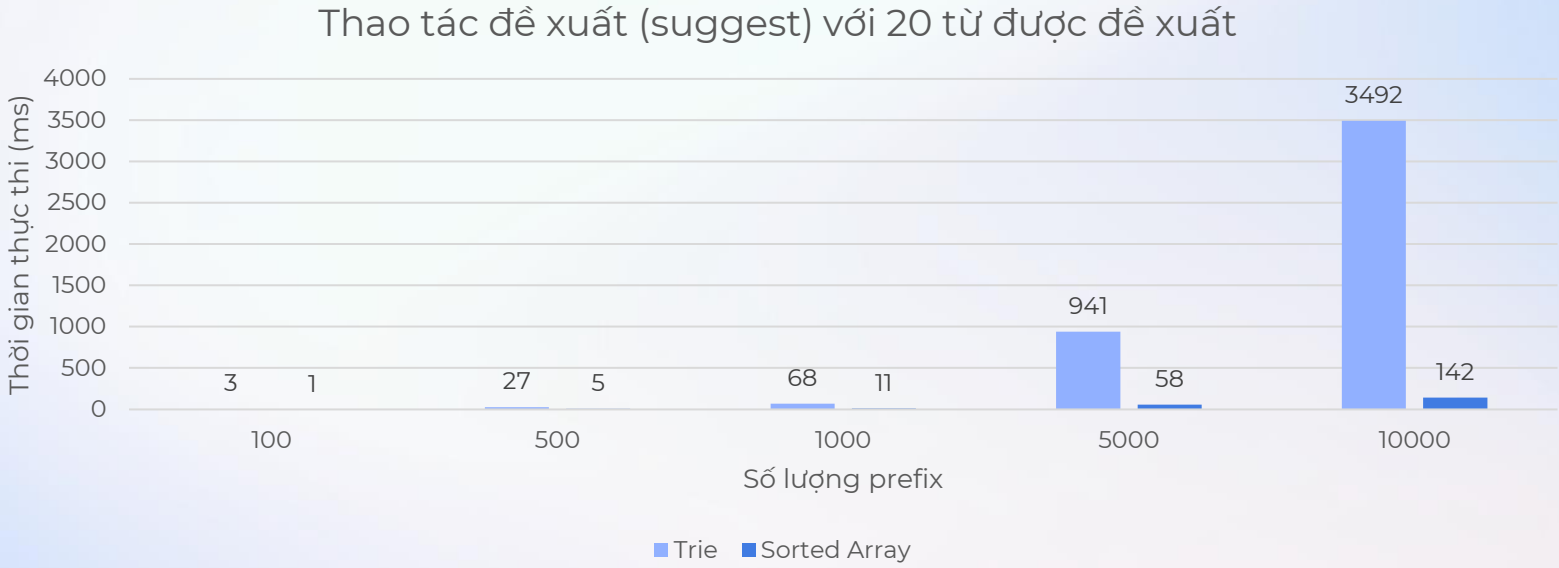
Với thao tác đề xuất (suggest), ở đây các prefix được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh thời gian thực hiện trung bình giữa Trie với Sorted Array như sau:



Đánh giá thuật toán

Số liệu thực nghiệm của thời gian thực thi

Với thao tác đề xuất (suggest), ở đây các prefix được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh thời gian thực hiện trung bình giữa Trie với Sorted Array như sau:



Đánh giá thuật toán

Kết luận

- Từ số liệu thực nghiệm trên, ta có thể dễ dàng thấy rằng quá trình **thêm hay xoá** một lượng lớn các từ vào Trie **nhanh hơn rất nhiều** so với Sorted Array. Lí do là vì các ô nhớ, đại diện cho 1 kí tự, chỉ được tạo ra và xoá đi khi cần thiết. Đồng thời việc tạo ra hay xoá đi này cũng không ảnh hưởng đến toàn bộ cấu trúc. Tuy nhiên đối với Sorted Array, việc tạo ra hay xoá đi một từ đồng nghĩa với việc phải di dời và phân bổ lại toàn bộ các phần tử trong mảng con (trong trường hợp xấu nhất là toàn bộ mảng). Do đó, 2 quá trình này tốn rất nhiều thời gian.

Đánh giá thuật toán

Kết luận

- Đối với quá trình **đề xuất từ**, Sorted Array đã thể hiện khả năng **vượt trội hơn** so với Trie. Lí do là vì việc đề xuất từ trong Sorted Array dựa trên thuật toán Binary Search để tìm kiếm vị trí bắt đầu và kết thúc của các từ bắt đầu prefix trong mảng. Quá trình này sẽ nhanh hơn và tiết kiệm bộ nhớ hơn là duyệt qua từng kí tự của từ trong Trie, vốn phải sử dụng nhiều stack cho kĩ thuật đệ quy và quay lui.

Đánh giá thuật toán

Kết luận

Như vậy, Trie thường vượt trội hơn khi thao tác với dữ liệu có tính di động, thường xuyên phải cập nhật. Còn Sorted Array vượt trội hơn khi thực hiện chức năng đề xuất lượng lớn từ với bộ dữ liệu cố định.

03. Tài liệu tham khảo

03. Tài liệu tham khảo

Tài liệu văn bản

[1] Knuth, Donald (1997). "6.3: Digital Searching". The Art of Computer Programming Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley. p. 492. ISBN 0-201-89685-0.

[2] Alan Jay Smith. 1982. Cache Memories. ACM Comput. Surv. 14, 3 (Sept. 1982), 473–530. https://doi.org/10.1145/356887.356892

[3] Baeza-Yates, R., & Navarro, G. (1998). *Fast approximate string matching in a dictionary*. Proceedings of SPIRE'98, IEEE CS Press, 14–22. http://www.dcc.uchile.cl/~gnavarro/ps/spire98.2.pdf

[4] В. И. Левенштейн (1965). Двоичные коды с исправлением выпадений, вставок и замещений символов [Binary codes capable of correcting deletions, insertions, and reversals]. Доклады Академии Наук СССР (in Russian). 163 (4): 845–848. Appeared in English as: Levenshtein, Vladimir I. (February 1966). "Binary codes capable of correcting deletions, insertions, and reversals". Soviet Physics Doklady. 10 (8): 707–710. https://ui.adsabs.harvard.edu/abs/1966SPhD...10..707L

[5] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). Introduction to automata theory, languages, and computation (3rd ed., pp. 83–122). Pearson.

03. Tài liệu tham khảo

Tài liệu ảnh

[6] Ethan Nam. (2019, February 27). Levenshtein distance formula [Image]. In Medium. Retrieved December 20, 2024, from https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0

fit@hcmus

Kết thúc phần trình bày!

Cảm ơn thầy đã theo dõi