

**SEARCH ENGINE**

Data Structure – CS163

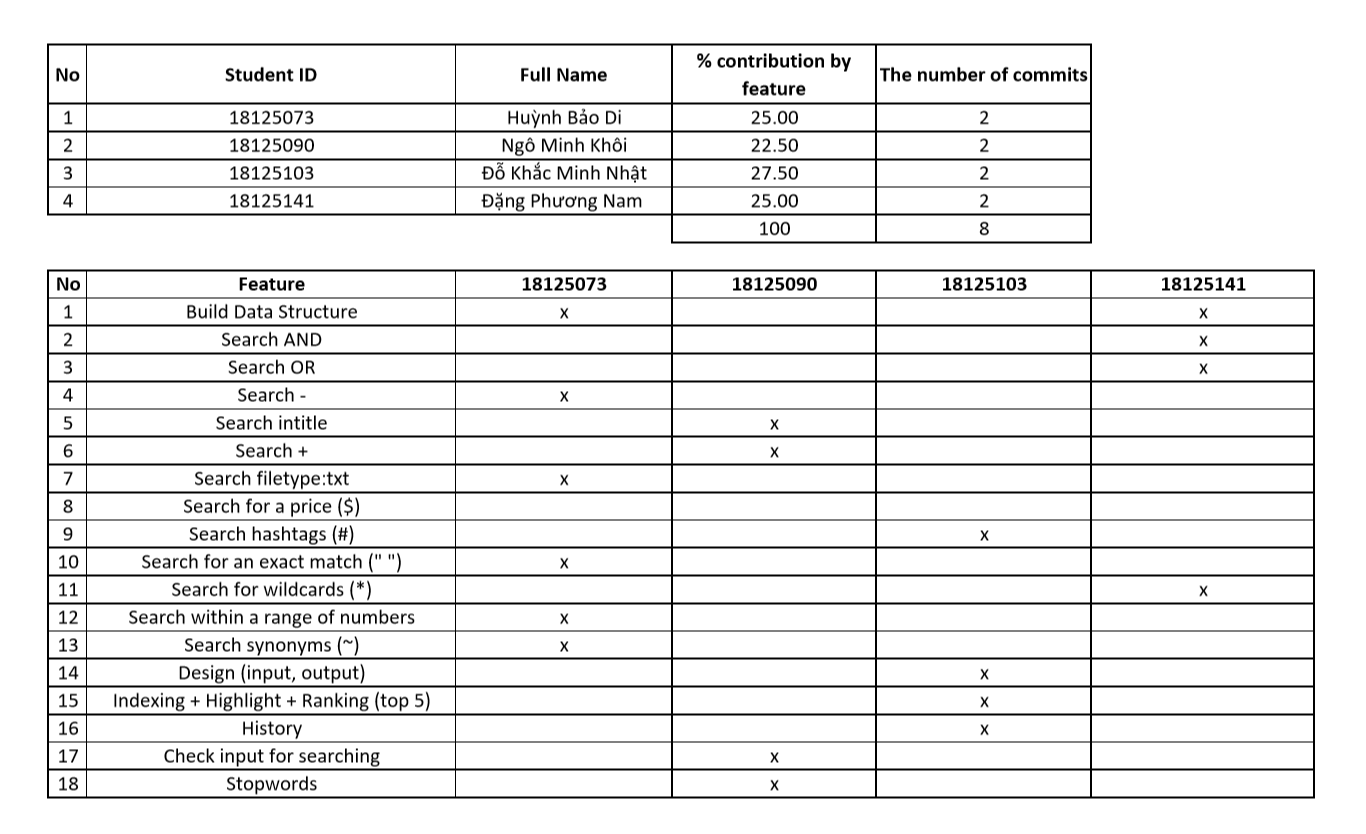
Project

Huỳnh Bảo Di – 18125073

Ngô Minh Khôi – 18125090

Đỗ Khắc Minh Nhật – 18125103

Đặng Phương Nam - 18125141

**BRIEF REPORT**

**1/ Design issues:**

* Implementation of data structure (suffix tree).
* Solution: we refer the sample document from several resource such as: GeekForGeeks, Youtube, etc.
* The out-of-memory problem. Because of the disavantages of suffix tree, our program takes a lot of spaces on RAM and make our computers run slower, stuck or even crash.
* Solution: reduce the number of characters inputing to the contruction per suffix tree, so that the memory consumption is also reduced. However, this solution cannot help contructing 11 000 suffix trees for 11 000 data files, the maximum number of files we can build is only 5200.

**2/ Data structure:**

* We use Suffix tree as a main data structure for the whole process.
* We also use Trie as a side data structure for storing the dictionary of synonym words.
* When we took our first step to researching for the project, after taking a look at a lot of difference type of trees, Suffix tree finally turn out to be the most suitable one because:
* It seems to be the fastest-searching data structure.
* It helps us easier to store the index of searching-words in order to highlight later.
* About Trie:
* It is definitely the best data structure for a normal dictionary!

**3/ Algorithms:**

* We use Ukkonen’s algorithms to construct a suffix tree.
* Because Ukkonen seems to be the easiest one to understand.
* …and some other common algorithms to do with the suffix tree: search, indexing, etc.

**4/ Particular emphasis:**

* The run-time to build a single suffixe tree and to search for a words among files of our program is quite fast.
* The problem is only the memory consumption like we have mentioned before.

**5/ Optimization issues:**

* Currently, we have not found out the method for further optimization. But if we keep working hard, it is only the matter of time.
* However, we think that our implementation can work efficiently with the huge collection of text, all we need is just more RAM, or stronger computer.

**. Run-time statistical analysis:**

**1/ Build suffix tree (affected by the condition of computer):**

* 1000 files: 20,5s - 24,25s
* 2000 files: 48,76s – 56,73s
* 4000 file: ~1min42s
* 5200 files: ~2p14s

**2/ queries searching:**

* **4000 files:**

1. John ~ 4s
2. ~bad ~ 7,14s
3. John Oliver ~ 0, 135s
4. “John Oliver” 0,006s
5. The cat ~ 2,64s
6. The dog ~ 1,86s
7. Mom AND dad ~ 1,75s
8. Mom OR dad ~ 0,975s
9. Singer ~ 2,21s
10. Computer science ~ 1,659s
11. Artificial intelligent ~ 1,879s
12. NASA ~ 1,037s
13. USA ~ 0,783s
14. Camera $400 ~ 0,453s
15. Camera $50..$400 ~ 2,26s
16. ~amazing ~ 7,257s
17. dad \* mom ~ 4,262s
18. John \* factory ~ 4,788s
19. Ho Chi Minh ~ 2,2s
20. BTS ~ 0,884s
21. “John Oliver” ~ 0.0012s
22. Vietnam ~ 1.26s
23. London ~ 1.12s
24. Paris ~ 1.22s
25. Hanoi ~ 1.21s

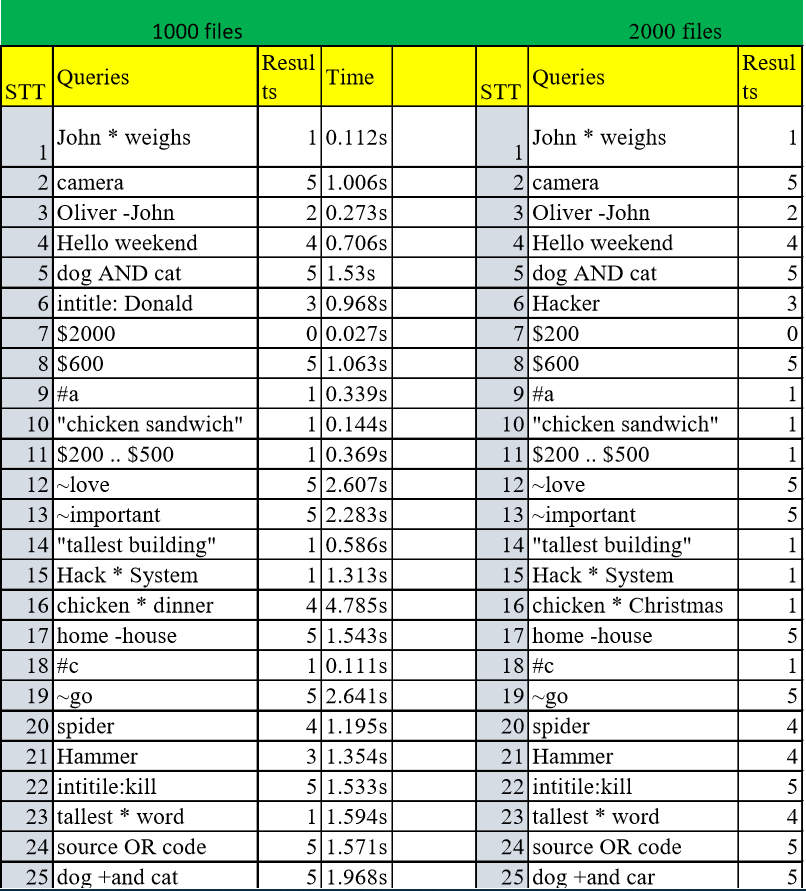
Etc.

* **5000 files:**

1. “John Oliver” ~ 3s
2. John ~ 2,41s
3. ~ bad ~ 7,008s
4. ~bad ~ 0,056s
5. John ~ 0,06s
6. “John Oliver” ~ 0,09s
7. Singer ~ 2,69s
8. The cat ~ 2,68s
9. Singer ~ 0,018s
10. Hanoi ~ 1.516s
11. London ~ 1.31s
12. Paris ~ 1.72s
13. Vietnam ~ 0.896s
14. Peter Parker ~ 0.309s
15. Programming ~ 3.012s
16. MSI ~ 1.179s
17. Hue ~ 0.19s
18. Handsome ~ 2.799s
19. Girl ~ 1.915s
20. Nhi ~ 0.916
21. Viet My ~ 0.017s
22. Bao Di ~ 1.082s
23. Museum ~ 1.55s
24. America ~ 0.99s
25. UK ~ 0.917s

Etc.

**The image below show the queries result for 1000 and 2000 files:**



Based on the above result, we can sumarize the run-time statistical analysis:

* The **fastest** run-time is **0.006s** and the **slowest** is **7,257s**
* As **clearer** the **input**, as **faster** the program **performs**
* **Synonym**, **unknown-words** and **in-range** searching have the **slowest** performance.
* **Exact-match** searching is the **fastest operator**.
* The program **performs faster** for the **input** as **popular words** such as: cities, celebrities, nations, etc.
* …and **slower** for **common** **daily** **words** such as: objects, verbs, animal, etc.
* When we **re-search** the word that has been searched before, the **run-time** seems to be **much faster** than the previous one.