Final Project Report

Team#11 (Option 3):
Chou, Kelvin, 862466579
Yu, Kunyi, 862548836
Ibrahim, Mona, 862267986

Abstract: This report focuses on the code and experiment replication of the provided paper [1] and corresponds to project option 3. It begins with a paper review, covering the summary of query problems, indexing architecture, and the POWER query processing algorithm. Next, the code replication process is introduced, including code decomposition and experiment design. Finally, the experiment results are presented and discussed, followed by the conclusion.

Index Terms: kNN spatial-keyword query, Indexing architecture, Query processing algorithm, Code replication

1. Paper Review

The paper of cutting-edge Spatial-keyword querying process is a 2022 published paper by Yongyi Liu and Amr Magdy [1]. Main contributions of the paper includes:

1) Propose two novel kNN spatial-keyword query problems
   - TKQN: Top-k kNN Query with Negative keyword predicates (we choose this)
   - BKQN: Boolean kNN Query with negative keyword predicates
2) Introduce U-ASK: a unified architecture for spatial-keyword query with negative keyword predictions
3) Introduce POWER: a query processing algorithm handling TKQN and BKQN queries
4) Present experimental evaluation on real datasets

Current work of the spatial-keyword querying process has two limitations: one is lack of dealing with phrases with a sequence of words, and the other one supports only boolean kNN query. To handle the limitations, U-ASK proposed by the paper supports two types of kNN spatial-keyword queries with AND, OR, and NOT conjunctions, which consists of an index framework TEQ (Textual Enhanced Quadtree) and a query processing algorithm POWER (Parallel Bottom-up Search with Incremental Pruning).

The problem of TKQN takes a tuple of location, positive/negative keywords, weight factor, and number k, then outputs top-k output with at least one positive keyword, without any negative keyword, and ranked by a spatial-textual-score function. The input format is as follows:

$$q_t = (q_t.loc, q_t.pos, q_t.neg, q_t.\lambda, q_t.k)$$

The score function is defined as:

$$score(o_i, q_t) = q_t.\lambda * score_s(o_i, q_t) + (1 - q_t.\lambda) * score_t(o_i, q_t)$$

Moreover, TEQ indexing is a hybrid, memory-resident index for TKQN queries which combines the strengths of a quadtree for spatial partitioning and an inverted index for keyword organization. Structure: each leaf cell n in the quadtree contains 4 components:

1) $n.ltp$: a location table pointer to a hash file on disk
2) $n.neigh$: a list of spatial neighboring cells
3) $n.iti$: an inverted textual index (hashtable) mapping keyword w to the tuple:
    a) $w.size$: number of objects containing the keyword w
    b) $w.max$: maximum weight of w in the cell
    c) $w.listPtr$: pointer to the sorted inverted list file
    d) $w.setPtr$: pointer to the sorted inverted set file (faster boolean filtering)
4) $n.oti$: an object textual index (hashtable) mapping object IDs to full text

The way of constructing TEQ contains two passes. The first pass builds the spatial indexing component: (build tree)

1) Insert objects into the quadtree
2) Quadtree cells are split based on object density
3) Create $n.neigh$ and $n.ltp$

The second pass builds the textual indexing component: (build cells)

1) Initialize two hashtables: $n.iti$ and $n.oti$
2) Insert every object o into $n.oti$
3) Insert every keyword w with its weight into $n.iti$
4) Sort inverted list and inverted set, store all parameters into disk

The query problem our group chose is TKQN, so as requirements, we need to implement the POWER algorithm. The POWER (Parallel bOttom-up search With incrEmental pRuning) operates within a master-worker framework, where each worker processes local top-k searches within assigned index cells. First, the POWER will process index cell by loading location table $n.LT$ into a buffer with LRU policy (prepared for parallelization, the project will omit it). Second, the POWER uses a top-k search strategy based on the Threshold Algorithm (TA), which incrementally retrieves and aggregates scores from multiple sorted lists. Thanks to the upper bound score, it will prune many useless searches. Third, because the TKQN query ranks its results based on textual attribute (can be found in keyword-inverted lists) and spatial attribute (sorted based on the query location), a priority queue will incrementally retrieve top-k objects based on the TA algorithm mentioned before. Lastly, the POWER will evaluate both positive keyword predictions ($q.pos$) and negative keyword predictions ($q.neg$) by a series rigorous steps. Note that, due to the computationally expensive nature of using a priority queue, the paper also introduces POWER-T (textual pruning) and POWER-S (spatial-pruning) to reduce query cost, but the project option 3 will not cover this part.

The experimental evaluation is in section 6 of the paper, which contains 6.1) experimental setup, 6.2) performance evaluation for the different parameters and framework components, and 6.3/4) compares the proposed algorithms against the SOTA under TKQN and BKQN, respectively. In addition to 6.1) experimental setup, the experiments suitable for this project are mainly distributed in 6.3) TKQN query evaluation.

For the 6.1) experimental setup, there is a summary in the "Table 2: Evaluation Parameters Values" of the paper.

For the 6.3) TKQN query evaluation, 4 experiments are conducted:

1) The effect of query keywords: change different number of positive words ($|q.pos|$), number of negative phrases ($|q.neg|$), and length of negative phrases ($q.negLen$), showing how the performance floats;
2) The effect of weighting factor: change different weighting factors (λ), showing how the balance between spatial and textual scores affects performance;

3) The effect of k: change different answer sizes ($q.k$), showing how the multi-threading master-worker architecture of POWER-T affects;
4) The effect of dataset size: change different dataset sizes, showing how the scalability varies.

The figures of above experiments' results can be found in the "Figure 5: TKQN Query Evaluation" of the paper.

2. Code Replication

In our code replication process, we choose Python as programming language and manage with multi files to decompose the complexity of the project. The python source files are stored in directory *./src* and the structure is as follows:

```
./src
exps
    exp_a.py
    exp_b.py
    exp_c.py  // deprecated, not applicable for POWER_batch
    exp_d.py  // deprecated, no lambda for POWER/POWER_batch
    exp_e.py
    exp_f.py
invertedIndex.py
main.py
plots.py
point.py
power.py
power_batch.py
quadTreeNode.py
query.py
read_data.py
```

The main entry of the project is *main.py*, which contains the main function to run the experiments. The *read_data.py*, *point.py*, *quadTreeNode.py*, *invertedIndex.py*, *query.py*, *power.py*, and *power_batch.py* are the core modules of the project. The implementation logic are follow the paper closely. Note that the *power_batch.py* is used to run the experiments in batch mode, which will first gether the queries with the same parameters into a group and then run the POWER algorithm for each group.

The *plots.py* is used to generate the plots for the experiments. The *exps* directory contains the experiments mentioned in the paper subsection 6.3. The python files in the *exps* directory will be called by the *main.py* to run the experiments.

Instruction to run the code:

```
$ python main.py --exp=<exp_name> --size=<dataset_size>
$ # exp_name: a, b, c, d, e, f (default a)
$ # dataset_size: 2, 4, 6, 8, 10 (default 4)
```

For the data input, we use the class provided one and rename it as *./data* directory. The data files are not included in the submission code due to the size limitation.

3. Experiment Results

The parameters settings for the experiments are remained the same as the paper. You may find those parameters in the *./src/exps/* directory and *./src/main.py* which are well commented. The following table shows the details which is copied from the paper[1], default settings use **bold** font format:

Parameter	Values
Dataset Size (million)	2, **4**, 6, 8, 10
Number of Threads	1, 2, **4**, 8
Buffer Size (MB)	50, 100, 150, **200**, 250, 300
$\|q.pos\|$	1 ,2 ,**3**, 4, 5
$\|q.neg\|$	**1**, 2, 3, 4, 5
$q.negLen$	1, **2**, 3, 4, 5
$q.\lambda$	0, 0.1, 0.3, **0.5**, 0.7 ,0.9 , 1
$q.k$ (TKQN)	5, **10**, 50, 100, 500, 1000, 5000, 10000
$q.k$ (BKQN)	**10**, 30, 50, 70, 90
$\|q.and\|$	1, **2**, 3, 4
$\|q.or\|$	1, **2**, 3, 4

Note that all experiments use 50 queries from the dataset.

In the paper, there are 6 experiments (a/b/c/d/e/f) conducted related to TKQN query evaluation. Among them, we have successfully replicated the experiments a, b, c, e, and f. Because the weighting factor λ is not presented in the POWER algorithm, experiment d is deprecated. The following parts will show the results of these experiments separately.

3.1. Experiment a

The purpose of experiment a is to evaluate the effect of changing the number of positive words to the performance of query latency (runtime).

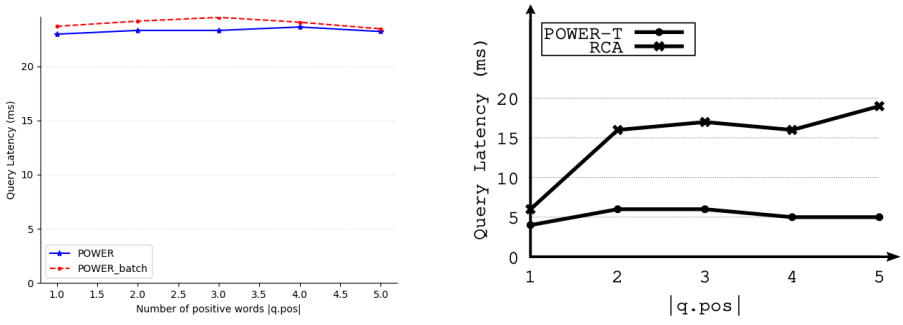

Fig. 1. Comparison of experiment a, left: our result, right: paper result

The left figure (our result) shows that the runtime of the POWER and POWER_batch algorithms barely changes (around 11ms) with different numbers of positive words. The result is similar to the right figure (paper result) where the query latency is stable at 5ms. The reason for the difference in the runtime maybe due to the different hardwares and implementation languages.

3.2. Experiment b

The purpose of experiment b is to evaluate the effect of changing the number of negative phrases to the performance of query latency (runtime).

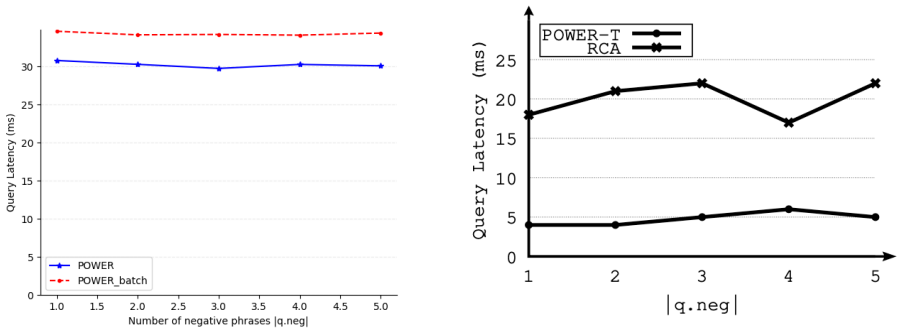

Fig. 2. Comparison of experiment b, left: our result, right: paper result

The left figure (our result) yields a great result that the runtime of both POWER and POWER_batch algorithms stay stable at around 33ms with different numbers of negative phrases. The batch version has a slightly higher runtime than the single version, may be due to the overhead of gathering queries. The right figure (paper result) shows that the runtime is stable at 5ms for all negative phrase numbers. The runtime difference may be due to the different hardwares and implementation languages.

3.3. Experiment c - Deprecated

The purpose of experiment c is to evaluate the effect of length of negative phrases to the performance of query latency (runtime). But the experiment c is not applicable for the POWER_batch algorithm.

3.4. Experiment d - Deprecated

The experiment d is deprecated because the weighting factor λ is not presented in the POWER algorithm.

3.5. Experiment e

The purpose of experiment e is to evaluate the effect of changing the parameter k (in kNN) to the performance of query latency (runtime).

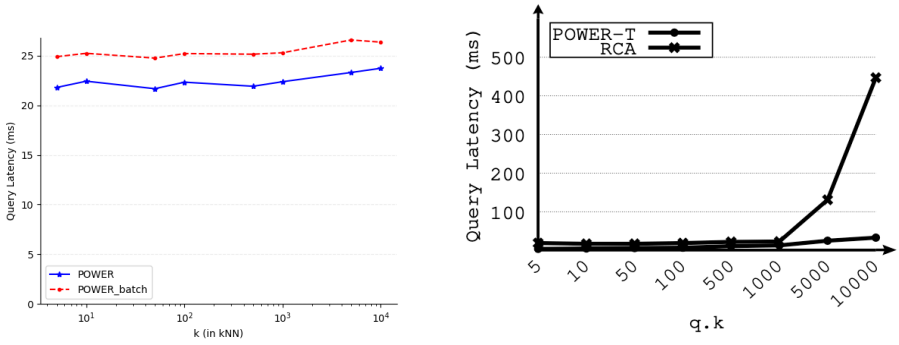

Fig. 3. Comparison of experiment e, left: our result, right: paper result

The left figure (our result) shows that the runtime of the POWER and POWER_batch algorithms remains stable at around 23ms with different values of k. At the same time the right figure (paper result) shows that the runtime is in a exponential growth trend with the increase of k. May be the reason for the difference is our implementation uses a small dataset size (4m) which only demonstrates the left part of the exponential growth.

3.6. Experiment f

The purpose of experiment f is to evaluate the effect of changing the dataset size to the performance of query latency (runtime).

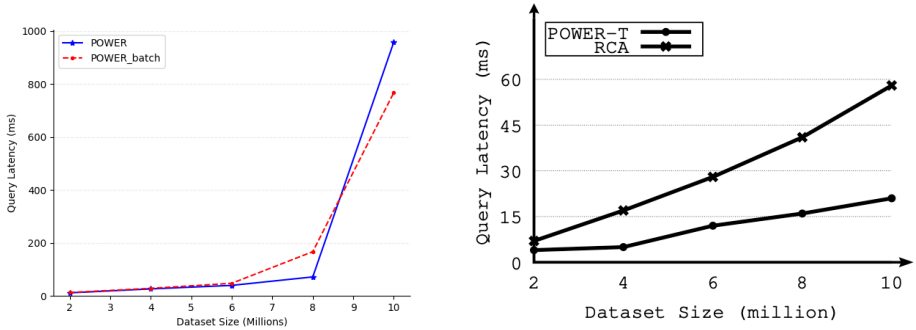

Fig. 4. Comparison of experiment f, left: our result, right: paper result

Comparing the left figure (our result) and the right figure (paper result), we can see that our version runtime is increasing exponentially with the dataset size while the paper version runtime is increasing linearly. We still need furthur investigation to find out the reason for this difference.

4. Conclusion

In this project, we have successfully replicated the experiments a, b, e, and f of the paper. The results of the experiments are similar to the paper results, but there are some differences in the runtime of the experiments. The differences may be due to the different hardwares and implementation languages, or a little bit implementation difference.

In this project, Mona and Kunyi have a tight cooperation from the understanding of the paper's content to the implementation and experiments. Kelvin also contributed to the paper reading and did his best to attempts the POWER_batch algorithm. The project is a good opportunity for us to learn the spatial computing knowledge and improve our programming skills. We hope to have more opportunities to learn and practice in the future.

Acknowledgements

The authors wish to thank Professor Amr Magdy, Dr. Yongyi Liu, TA Alhassan Satii Alshareedah, and others who have helped us in this course for their valuable suggestions and patient guidance. Wish them all the best in their future work and life!

The contribution of each team member is as follows:

- Chou, Kelvin
  - Paper reading
  - final report (appendix part)
  - README.md
  - Attempts of power_batch.py (in ./Kelvin's src)
- Yu, Kunyi
  - Paper reading
  - Write the assignment3/5, final report
  - Code decomposition
  - Experiments implementation of a/b/e/f (provide 4 images of section 3)
- Ibrahim, Mona
  - Paper reading
  - Implemented the whole source code
    - point.py
    - quadTreeNode.py
    - invertedIndex.py
    - power.py
    - power_batch.py
    - read_data.py
    - structured the queries
  - Experiments implementation of a/b/c (provide queries)

References

[1] Liu, Yongyi, and Amr Magdy. "U-ASK: a unified architecture for kNN spatial-keyword queries supporting negative keyword predicates." Proceedings of the 30th International Conference on Advances in Geographic Information Systems. 2022.

Appendix

The following part is Kelvin's demonstration of his unfinished implementation for TKQN queries using the POWER algorithm:

Kelvin's Unfinished Implementation for TKQN Queries using POWER Algorithm

Dataset Preparation

The dataset comprises spatial objects defined by their coordinates and associated keywords. Data preprocessing utilized Python's Pandas library for structured handling:

```
id,x,y,keywords
1,10,20,restaurant cafe
2,15,25,cafe bakery
3,20,30,restaurant bakery
```

POWER Algorithm Implementation

Due to constraints of the standard `rtree` Python library, a simplified brute-force method was implemented. This method evaluates keyword conditions and distances between query points and dataset objects directly:

```
def satisfies_keywords(keywords, positive, negative):
return positive.issubset(keywords) and negative.isdisjoint(keywords)

def distance(p1, p2):
return ((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)**0.5

def POWER_simple(query_location, positive, negative, k, data_objects):
results = []
for obj in data_objects:
if satisfies_keywords(obj['keywords'], positive, negative):
dist = distance(query_location, obj['location'])
heapq.heappush(results, (dist, obj))
return heapq.nsmallest(k, results)
```

Batch Query Processing

Batch query processing reduces redundant computations by executing multiple queries simultaneously:

```
def batch_POWER_simple(query_set, k, data_objects):
batch_results = {}
for qid, query in enumerate(query_set):
q_loc, positive, negative = query
results = POWER_simple(q_loc, positive, negative, k, data_objects)
batch_results[qid] = results
return batch_results
```