CHAPTER III

MINING TOP-*K* REGULAR-FREQUENT ITEMSETS

Based on the idea of "Controlling the number of regular-frequent itemsets to be mined"motivated from (Fu et al., 2000) and (Tanbeer et al., 2009), a problem of mining k regular-frequent with highest supports is introduced and defined in this chapter. Besides, an efficient single-pass algorithm named *Mining Top-K Periodic(Regular)-frequent Patterns (MTKPP)*, used to mine this kind of itemsets is also presented. To discover a set of top-*k* regular-frequent itemsets, the users can specify only a regularity threshold and a number of desired results instead of setting a support threshold. By avoiding the setting of a support threshold, this approach might help the users from the difficulty of specifying an appropriate support threshold to mine regular-frequent itemsets.

3.1 Top-*k* regular-frequent itemsets mining

This section introduces the basic notations and definitions needed to define top-*k* regular-frequent itemsets as defined in (Amphawan et al., 2009).

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of $n \geq 1$ literals, called *items*. A set $X = \{i_{j_1}, \ldots, i_{j_l}\} \subseteq I$ is called an *itemset* or an *l-itemset* (an itemset of size l). A transactional database $TDB = \{t_1, t_2, \ldots, t_m\}$ is a set of transactions in which each transaction $t_q = (q, Y)$ is a tuple containing unique transaction identifier q (tid in the latter) and an itemset Y. If $X \subseteq Y$, it is said that t_q contains X (or X occurs in t_q) and is denoted as t_q^X. Therefore, $T^X = \{t_p^X, \ldots, t_q^X\}$, where $1 \leq p \leq q \leq |TDB|$, is the set of all ordered tids (called *tidset*) where X occurs. The support of an itemset X, denoted as $s^X = |T^X|$, is the number of tids (transactions) in TDB where X appears.

Definition 3.1 (Regularity of an itemset X) *Let t_j^X and t_k^X be two consecutive tids in the tidset T^X of an itemset X,* i.e. *where $j < k$ and there is no transaction t_i, $j < i < k$, such that t_i contains X. Then, $rtt^X = t_k^X - t_j^X$ is the regularity value which represents the number of transactions not containing X between two consecutive transactions t_j^X and t_k^X. Thus, $RTT^X = \{rtt_1^X, rtt_2^X, \ldots, rtt_{m+1}^X\}$ is denoted as the set of all regularities of X. Then, the regularity of X can be defined as*

$$r^X = max(RTT^X) = max(rtt_1^X, rtt_2^X, \ldots, rtt_{m+1}^X)$$

Definition 3.2 (Regular-frequent itemset) *An itemset X is called a* regular-frequent itemset *if (i) its regularity is no greater than a user-given regularity threshold (σ_r); (ii) its support is no less than a user-given support threshold (σ_s).*

Thus, the regular-frequent itemsets mining problem is to discover a complete set of regular-frequent itemsets from transactional database with two user-given support and regularity thresholds. However, as mentioned in the previous chapter the user may prefer to specify a simple threshold on the amount of results instead of a support threshold. The following definition of a *top-k* regular-frequent itemsets mining problem is thus proposed.

Definition 3.3 (Top-*k* regular-frequent itemset) *An itemset X is called a* top-k regular-frequent itemset *if (i) its regularity is no greater than a user-given regularity threshold (denoted as σ_r) and (ii) there exist no more than $k-1$ itemsets whose their supports are higher than that of X.*

Therefore, the top-*k* regular-frequent itemsets mining problem is to discover a set of top-*k* regular-frequent itemsets from transactional database with two user-given parameters: a number k of expected outputs and a regularity threshold σ_r.

3.2 Preliminary of MTKPP

In this section, details of the MTKPP algorithm which is an efficient single-pass algorithm used to discover a set of k regular itemsets with highest supports from a transactional database are introduced. It adopts a best-first search strategy to quickly find regular itemsets with the highest values of support. MTKPP is based on the use of a top-*k* list (with hash table) structure to maintain top-*k* regular-frequent itemsets during mining process.

3.3 MTKPP: Top-*k* list structure

Top-*k* list is a linked-list used to maintain *k* periodic(regular)-frequent patterns with highest supports. A hash table is also used with the top-*k* list in order to quickly access information in the top-*k* list. At any time during mining process, the top-*k* list contains not much more than k regular-frequent itemsets in main memory. Each entry in a top-*k* list consists of 4 fields: an item or itemset name (I), a total support (s^I), a regularity (periodicity) (r^I) and a tidset where I occurs (T^I). For example in Figure 3.1, an item a has a support of 8, a regularity of 3. Its tidset is $\{1, 4, 6, 7, 8, 10, 11, 12\}$ which means the item a occurs in $\{t_1, t_4, t_6, t_7, t_8, t_{10}, t_{11}, t_{12}\}$.

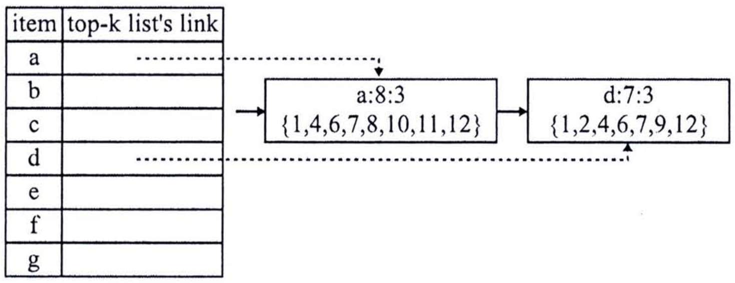


Figure 3.1: MTKPP: Top-*k* list with hash table

3.4 MTKPP algorithm

MTKPP consists of two steps: (*i*) Top-*k* list initialization: scan a database once to obtain k regular items (with highest support) and collect them into the top-*k* list with their supports, regularities and tidsets; and (*ii*) Top-*k* mining: merge each pair of entries in the top-*k* list by using the best-first search strategy (*i.e.* finding the itemsets with the highest support first in order to reduce search space) to generate a larger candidate itemset and then sequentially intersect their tidsets to calculate support and regularity of the new generated itemset.

3.4.1 MTKPP: Top-*k* list initialization

To create the top-*k* list, the database is scanned once to obtain all items. At the first occurrence of each item, the MTKPP algorithm creates a new entry in the top-*k* list and then initializes its support, regularity and tidset. For other occurrences, the hash table is looked up to find the existing entry in the top-*k* list and update the entry values. All items that have regularity greater than σ_r are removed from the top-*k* list and the top-*k* list is sorted in support descending order. Finally, all items that have support less than the support of the k^{th} item in top-*k* list (s_k) are removed from the top-*k* list. The details of the top-*k* list initialization process are described in Algorithm 1.

3.4.2 MTKPP: Top-*k* mining

To mine a set of top-*k* regular-frequent itemsets from the top-*k* list, the best-first search strategy is adopted first to generate regular itemsets with the highest supports. To generate a new candidate itemset, MTKPP starts from considering the most regular-frequent item to the least regular-frequent item in the top-*k* list. It then combines two entries in the top-*k* list under the following two constraints: (i) the size of the itemsets of both considered entries must be equal; (ii) both itemsets must have the same prefix (*i.e.* each item from both itemsets is the same, except the last item). When both itemsets satisfy the two constraints above, MTKPP will sequentially intersect their tidsets in order to calculate the support, the regularity, and the tidset of the new

Algorithm 1 (MTKPP: Top-*k* list initialization)

```
Input:
    (1) A transaction database: $TDB$
    (2) A number of itemsets to be mined: $k$
    (3) A regularity threshold: $\sigma_r$
Output:
    (1) A top-k list

    create a hash table for all 1-items
    for each transaction $j$ in $TDB$ do
        for each item $i$ in the transaction $j$ do
            if the item $i$ does not have an entry in the top-k list then
                create a new entry for the item $i$ with $s^i = 1, r^i = t_j$ and create a tidset $T^i$ that contains $t_j$
                create a link between the hash table and the new entry
            else
                add the support $s^i$ by 1
                calculate the regularity $r^i$ by $t_j$
                collect $t_j$ as the last tid in $T^i$

    for each item $i$ in the top-k list do
        calculate the regularity $r^i$ by $|TDB|-$ the last tid of $T^i$
        if $r^i > \sigma_r$ then
            remove the entry $i$ out of the top-k list

    sort the top-k list by support descending order
    remove all of entries after the $k^{th}$ entry in the top-k list
```

Algorithm 2 (MTKPP: Top-*k* mining)

```
Input:
    (1) A top-k list
    (2) A number of itemsets to be mined: $k$
    (3) A regularity threshold: $\sigma_r$
Output:
    (1) A set of top-k regular-frequent itemsets

    for each entry $x$ in the top-k list do
        for each entry $y$ in the top-k list $(x > y)$ do
            if the entries $x$ and $y$ have the same size of itemsets and the same prefix then
                merge the itemsets of $x$ and $y$ to be itemset $Z = I^x \cup I^y$
                for each $t_p$ in $T^{I^x}$ and $t_q$ in $T^{I^y}$ do
                    if $t_p = t_q$ then
                        calculate the regularity $r^Z$ by $t_p$
                        add the support $s^Z$ by 1
                        collect $t_p$ as the last tid in $T^Z$
                calculate the regularity $r^Z$ by $|TDB|-$ the last tid of $T^Z$
                if $r^Z \leq \sigma_r$ and $s^Z \geq s_k$ then
                    remove the $k^{th}$ entry from the top-k list
                    insert the itemset $Z$ $(I^x \cup I^y)$ into the top-k list with $r^Z$, $s^Z$ and $T^Z$
```

generated candidate itemset. If the regularity of the new candidate itemset is not greater than σ_r and the support is greater than the support of the k^{th} regular itemset in the top-*k* list, then the k^{th} regular itemset will be removed from the top-*k* list and the newly generated candidate itemset is inserted into the top-*k* list. The details of the mining process are described in Algorithm 2.

3.5 Example of MTKPP

Let consider the TDB presented in Table 3.1. The regularity threshold σ_r and the number of required results k are 4 and 5, respectively. Figure 3.2 illustrates the creating of the top-*k* list process from the TDB.

Table 3.1: A transactional database as a running example of MTKPP

tid	items
1	*a b d e*
2	*c d e*
3	*b c f g*
4	*a b d f g*
5	*c e g*
6	*a b c d g*
7	*a b c d*
8	*a b c e*
9	*b c d*
10	*a c e g*
11	*a b f*
12	*a b d g*

With the scanning of the first transaction $t_1 = \{a, b, d, e\}$, the entries for items a, b, d and e are initialized in the top-*k* list as shown in Figure 3.2(a). The next transaction ($t_2 = \{c, d, e\}$) initializes a new entry in the top-*k* list for item c. It then updates the values of support and regularity for items d and e to be $2 : 1$ and their tidsets to be $\{1, 2\}$ (Figure 3.2(b)). As shown in Figure 3.2(c), after scanning the third transaction ($t_3 = \{b, c, f, g\}$), the regularity r^b of the item b changes from 1 to 2. The top-*k* list after scanning all transactions is given in Figure 3.2(d). Next, the item f which has the regularity $r^f = 7$ greater than $\sigma_r = 4$ is removed from the top-*k* list. Finally, the top-*k* list is sorted by support descending order and item e is removed from the top-*k* list, since the support of e ($s^e = 5$) is less than support of g ($s^g = 6$) which is the $k^{th}(5^{th})$ pattern in the top-*k* list. The top-*k* list after initialization phase is shown in Figure 3.2(e).

MTKPP mines the top-*k* regular-frequent itemsets from the top-*k* list of Figure 3.2(e). Since item b is the first item in the top-*k* list and it has no items in the previous sequence, MTKPP starts by considering item a and search for identical size and prefix items (in the previous sequence), item b. Then, item b is combined with item a and their tidsets are intersected to find the support ($s^{ba} = 7$), the regularity ($r^{ba} = 3$) and the tidset ($T^{ba} = \{1, 4, 6, 7, 8, 11, 12\}$) of itemset ba. Since the regularity of ba is less than $\sigma_r = 4$ and the support of ba is more than $s_k = 6$, the itemset ba is inserted into the top-*k* list and item g (the k^{th} itemset) is removed from the top-*k* list (Figure 3.3). Next, the third element, item *c*, is considered. There are two entries which are in the previous sequence and have the same prefix as *c*: b and a. Thus, item *c* is combined with

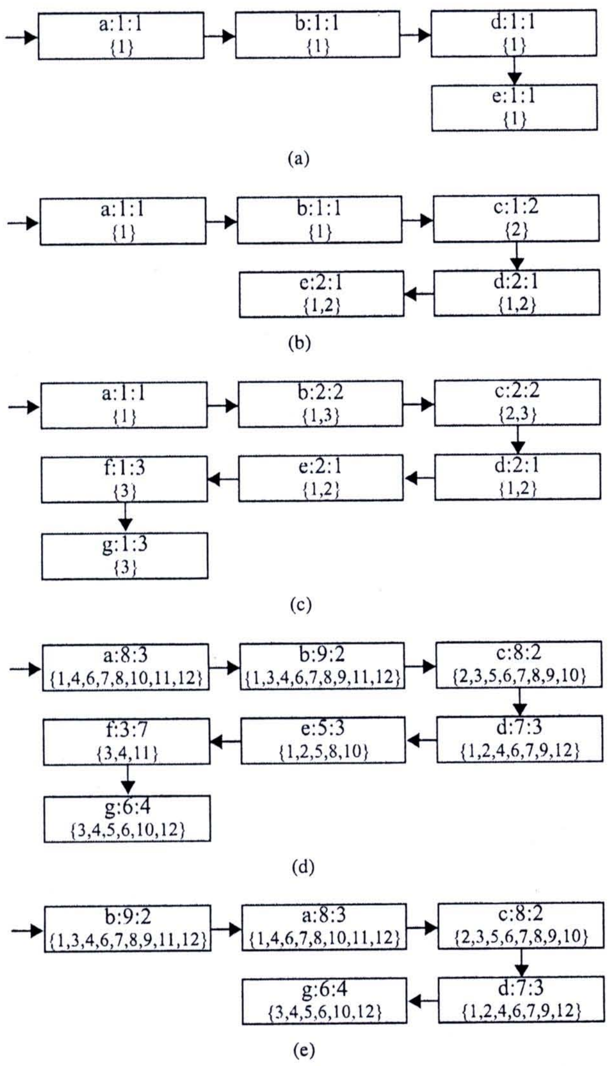


Figure 3.2: Top-*k* list initialization

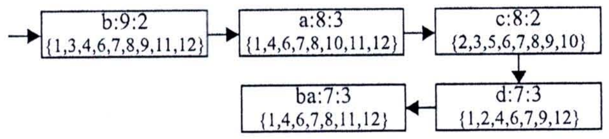


Figure 3.3: Top-*k* regular-frequent itemsets

item b and their tidsets are intersected. The tidset and the regularity of cb are $\{3, 6, 7, 8, 9\}$ and 3, respectively. Because the support of cb ($s^{cb} = 5$) is less than the support of $s_k = 7$, the itemset cb is no longer considered. Next, item c and item a are combined and their tidsets are intersected. The tidset of ca is then $\{6, 7, 8, 10\}$. Since the regularity of $ca(r^{ca} = 6)$ is greater than 4, itemset ca cannot be a regular itemset. Next, item d and itemset ba are considered in the same manner. When all itemsets in the top-k list have been considered, the top-k regular-frequent itemsets are stored in the top-k list with their occurrence information. The final result is shown in Figure 3.3.

3.6 Performance evaluation

In this section, the experimental studies are reported in order to evaluate the performance of the MTKPP algorithm. From the best of our knowledge, there is no other existing approach to discover top-*k* regular-frequent itemsets. Then, the effectiveness of MTKPP algorithm is focused and compared with PF-tree (Tanbeer et al., 2009) which is a regular-frequent itemset mining algorithm. It should be noticed that PF-tree mines the regular-frequent itemsets with a user-given support threshold whereas MTKPP requires the number of regular-frequent itemsets to be mined (k). Then, the support threshold is fixed in the way that PF-tree mines the same set of regular-frequent itemsets with highest supports as MTKPP (*i.e.* it is specified as $\sigma_s = s_k$ which is equal to the lowest support of the set of top-*k* regular-frequent itemsets). To demonstrate the performance of MTKPP, the processing time (*i.e.* CPU and I/Os costs) is investigated to compare the performance of the two algorithms with the small and large values of k and various values of regularity threshold (σ_r). Furthermore, a study of memory consumption of MTKPP is also considered because of the use of the top-*k* list structure. Lastly, the scalability of MTKPP on the number of transactions in the database is evaluated.

3.6.1 Experimental setup

As shown the characteristics in Chapter 2, nine real (*i.e.* accidents, BMS-POS, chess, connect, kosarak, mushroom, pumsb, pumsb*, retail) and three synthetic (*i.e.* T10I4D100K, T20I6D100K, and T40I10D100K) datasets were employed to examine the performance of MTKPP. The simulations were performed on a Intel®Xeon 2.33 GHz and with 4 GB main memory on a Linux platform and the program of MTKPP and PF-tree implemented in C. In the experiments, the value of σ_r is set depending on the characteristic of each dataset for illustrative purpose. Therefore, the value of σ_r is specified to be different values. In fact, the number of regular itemsets for each database increases with the value of the regularity threshold. On sparse datasets, each itemset does not occur frequently thus the value of σ_r should be set to be large when the value of k is large. While, each itemset appears very often in dense dataset, a small value of σ_r should be applied. Hence, the value of k is divided into two rages: (*i*) [50,500] for the small values; and (*ii*) [1,000, 10,000] the large values, respectively.

3.6.2 Execution time

Figure 3.4 to Figure 3.21 show the runtime of MTKPP and PF-tree on real dense datasets (*i.e.* accidents, chess, connect, mushroom, pumsb, and pumsb*). From these figures, it can be observed that in almost cases, MTKPP outperforms PF-tree with the small and large values of

k. However, in some cases especially on connect and mushroom datasets when the value of k is large, MTKPP cannot significantly reduce the computational time from PF-tree. This happen because these two datasets have a small number of transactions (in some cases the number of transaction is less that the number of desired results). Then, PF-tree can reduce time to merge tidset from children to parent nodes, while MTKPP cannot take the advantage of using a top-k list.

Figure 3.22 to Figure 3.36 illustrate the processing time of two real sparse datasets (*i.e.* BMS-POS and retail) and the three synthetic datasets (T10I4D100K, T20I6D100K and T40I10D100K). One can observe that the computation time of MTKPP increases as k increases. When the value of k increases, MTKPP has to find more results, therefore the computation time increases as well. By comparing with PF-tree, MTKPP can save a large amount of time for small and large value of k. MTKPP runs very fast on sparse datasets since each itemset occur rarely (*i.e.* the number of tids that each itemset occurs is few). As a result, MTKPP spent a little time to intersect tidsets while PF-tree take time to merge and order tids. Therefore, these results confirm the advantage of MTKPP over PF-tree for the real and synthetic sparse datasets where the item distributes not regularly.

3.6.3 Memory consumption

The variation of memory usage of MTKPP with the number of regular-frequent itemsets to be mined, k, is shown in Figure 3.37 to Figure 3.47.

From these figures, it is obvious that the memory usage increases as k increases. In fact, the desired memory of MTKPP depends on the support of each itemset in the top-k list because MTKPP has to maintain the tidsets (*i.e.* sets of tids) of all itemsets in the top-k list in order to calculate their support and the regularity. For dense datasets, the memory usage linearly increases because the supports of itemsets in the top-k list are very close. For sparse datasets, the memory usage increases slightly as k increases because the supports of itemsets in the top-k list are quite different. However, based on the used of the top-k list structure, the memory usage of MTKPP is efficient for the top-k regular-frequent itemsets mining using the recently available gigabyte range memory.

3.6.4 Scalability test

The scalability of MTKPP algorithm is also studied on execution time and memory consumption by varying the number of transactions in database. The kosarak dataset is used to test scalability with the number of transactions. The kosarak dataset is a huge dataset with a large number of distinct of items $(41,270)$ and transactions $(990,002)$. First, the database was divided into six portions (*i.e.* $100K$, $200K$, $400K$, $600K$, $800K$ and $990K$ transactions). Then, the performance of MTKPP was investigated on each portion. Second, the value of k is specified to be 500 and $10,000$ to investigate the scalability on the small and the large values of k. The regularity threshold was fixed to 6% of the number of transactions in each portion.

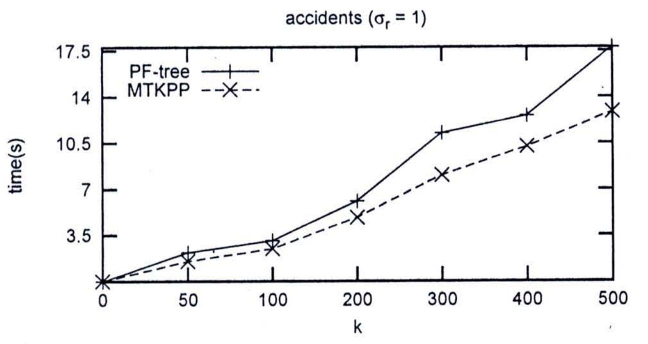
The experimental results shown in Figures 3.48 and 3.49. It is clear from the graphs that as the database size increases, overall top-*k* list initialization time and top-*k* mining time are linearly increased. The performance between MTKPP and PF-tree is similar when the number of transactions is between 0 and $200K$ transactions. Besides, MTKPP runs faster than PF-tree with the large number of transactions for the small and the large values of k. As shown the memory consumption of MTKPP in the figures, the memory requirement increases as the database size increases. However, MTKPP shows stable performance of about linearly increase of the runtime and memory usage with respect to the database size. Therefore, it can be observed from the scalability test that MTKPP can mine the top-*k* regular-frequent patterns over large datasets and distinct items with considerable amount of runtime and memory.

3.7 Summary

This chapter introduced and studied the problem of mining the top-*k* regular (periodic)-frequent itemsets from transactional databases without setting a support threshold. This problem allows users to control (or specify) the number of regular itemsets (*i.e.* the regularly-occurred itemsets) to be mined.

To discover this kind of itemset, an efficient one-pass algorithm, called *MTKPP (Mining Top-*K *Periodic(Regular)-frequent Patterns)*, is presented. Since the minimum support to retrieve top-*k* regular-frequent itemsets cannot be known in advance, a new best-first search strategy is devised to efficiently retrieve the top-*k* regular-frequent itemsets and the intersection process is applied to compute the support and the regularity of each itemset. By using these techniques, MTKPP first considers the itemsets with the highest support and then combines candidates to build the top-*k* regular-frequent itemsets list.

In the experiments, the empirical studies on both real and synthetic data (with the small and large values of k) show that the MTKPP algorithm is efficient for top-*k* regular-frequent itemset mining. It is also linearly scalable with the number of transactions comparing with PF-tree.



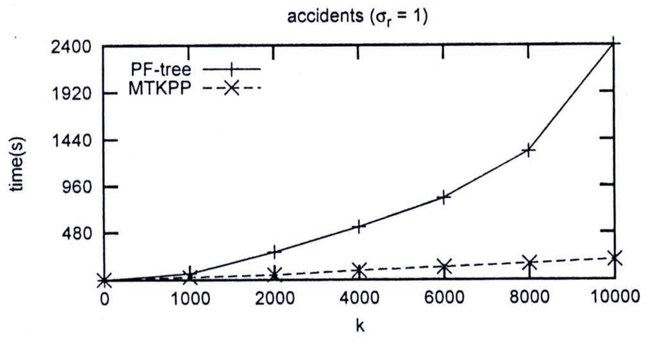
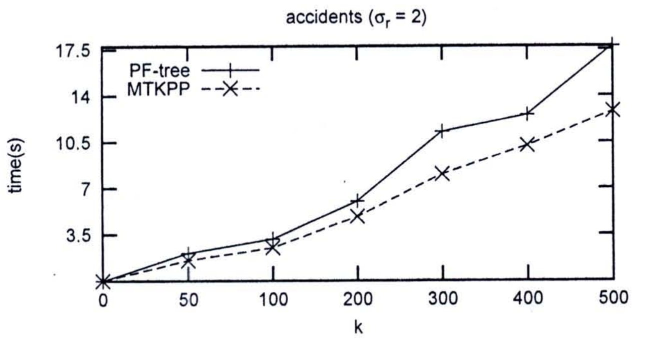


Figure 3.4: Runtime of MTKPP on *accidents* ($\sigma_r = 1\%$)



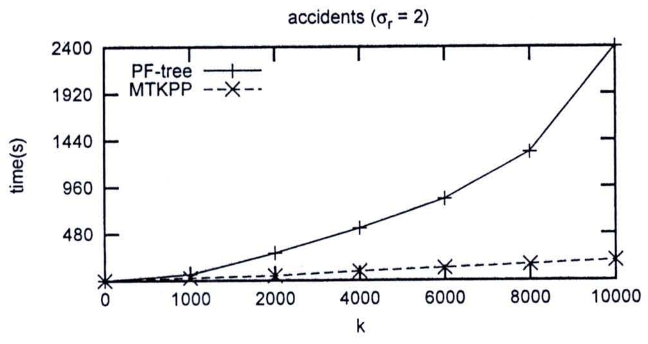


Figure 3.5: Runtime of MTKPP on *accidents* ($\sigma_r = 2\%$)

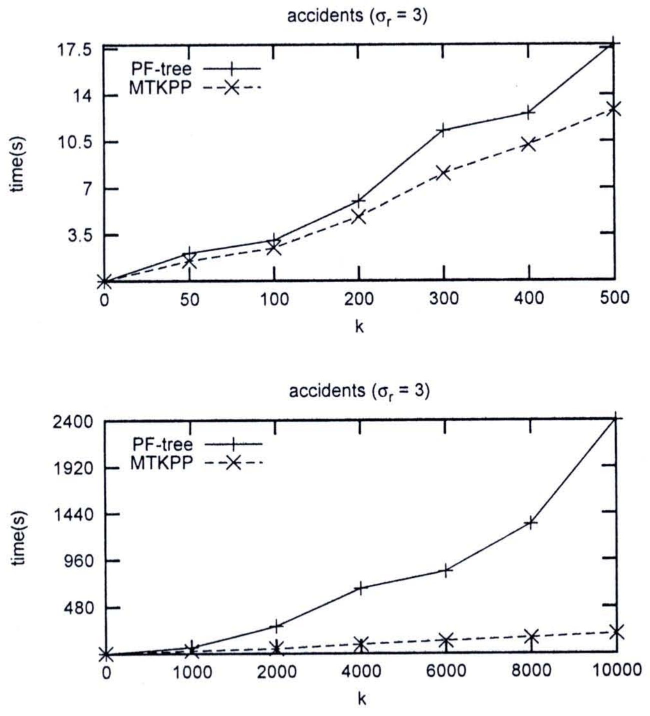


Figure 3.6: Runtime of MTKPP on *accidents* ($\sigma_r = 3\%$)

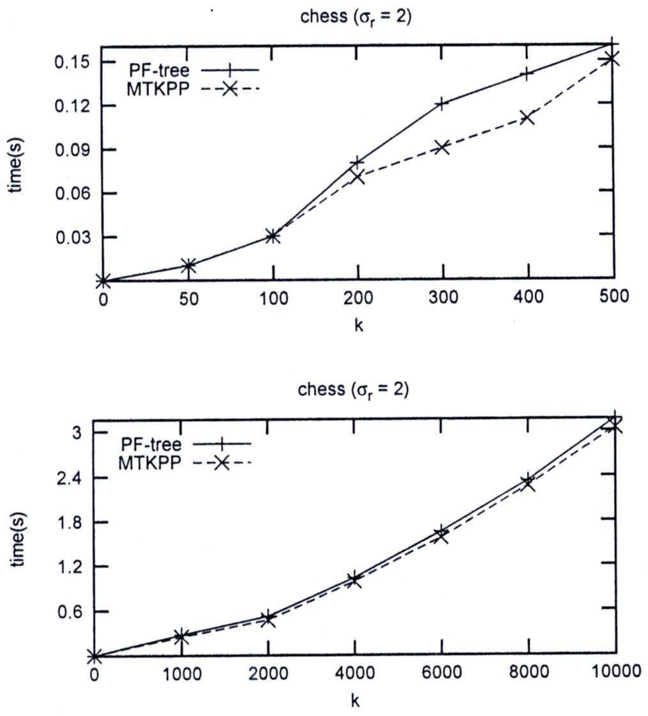


Figure 3.7: Runtime of MTKPP on *chess* ($\sigma_r = 2\%$)

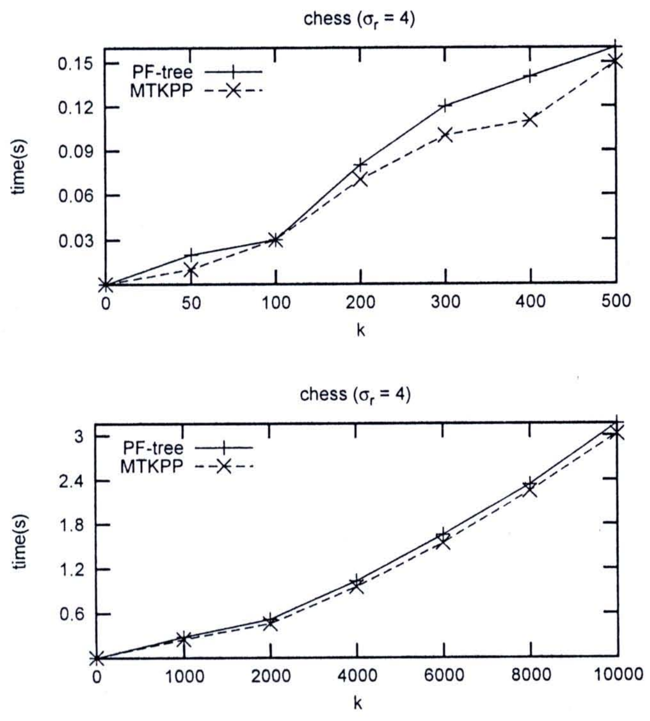


Figure 3.8: Runtime of MTKPP on *chess* ($\sigma_r = 4\%$)

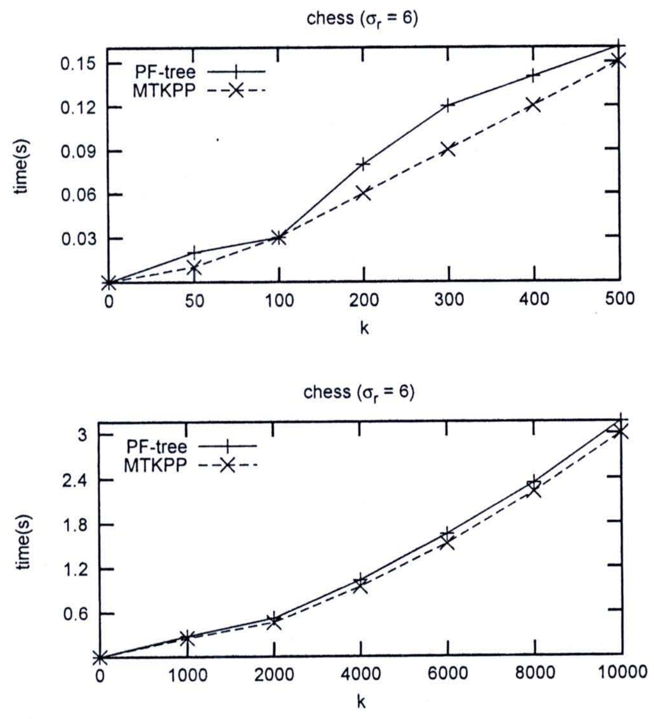


Figure 3.9: Runtime of MTKPP on *chess* ($\sigma_r = 6\%$)

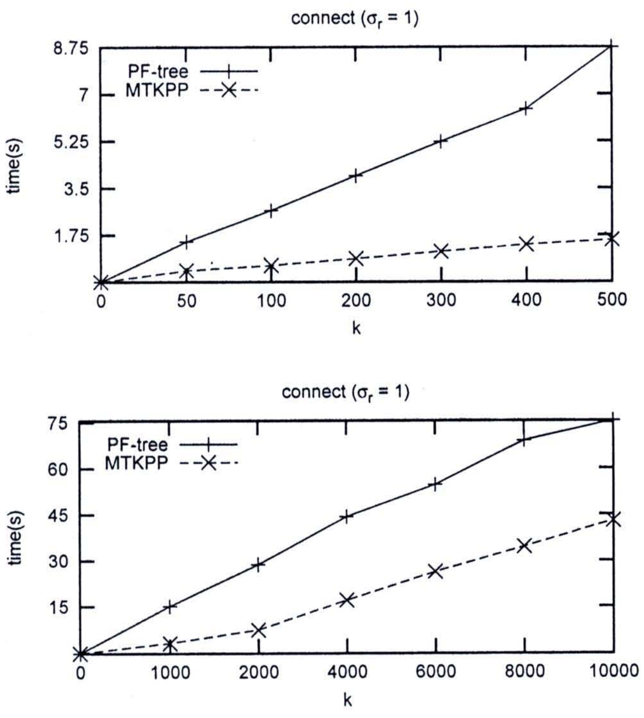


Figure 3.10: Runtime of MTKPP on *connect* ($\sigma_r = 1\%$)

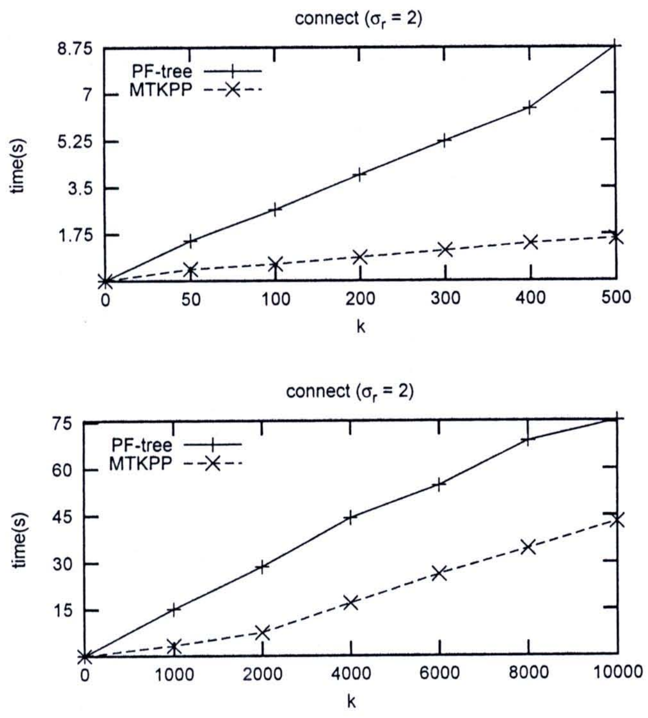
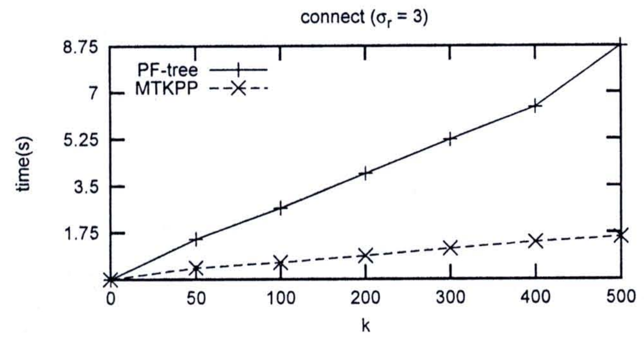


Figure 3.11: Runtime of MTKPP on *connect* ($\sigma_r = 2\%$)



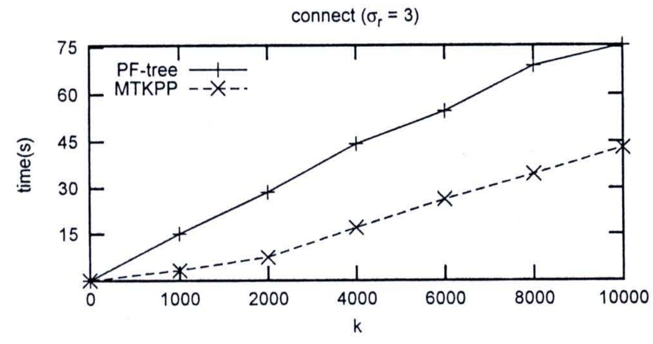
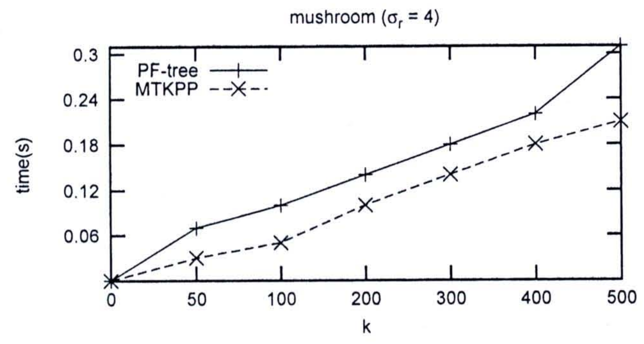


Figure 3.12: Runtime of MTKPP on *connect* ($\sigma_r = 3\%$)



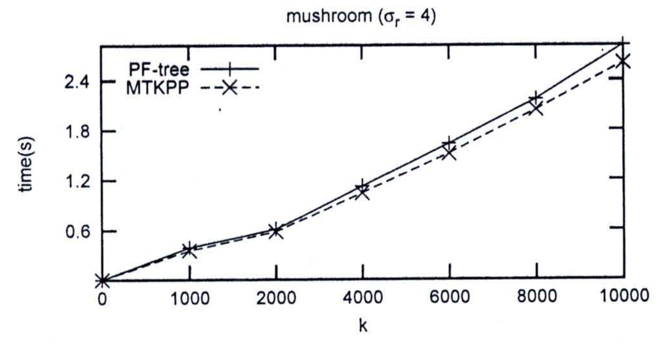


Figure 3.13: Runtime of MTKPP on *mushroom* ($\sigma_r = 4\%$)

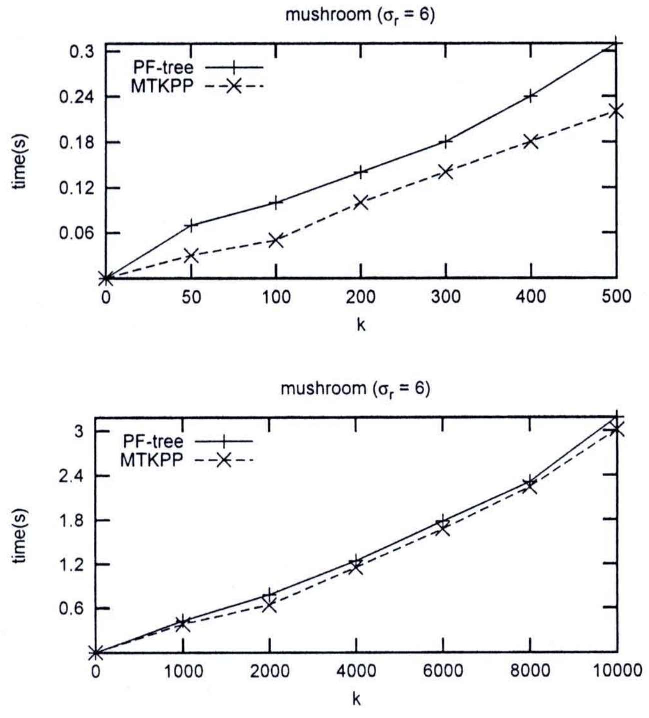


Figure 3.14: Runtime of MTKPP on *mushroom* ($\sigma_r = 6\%$)

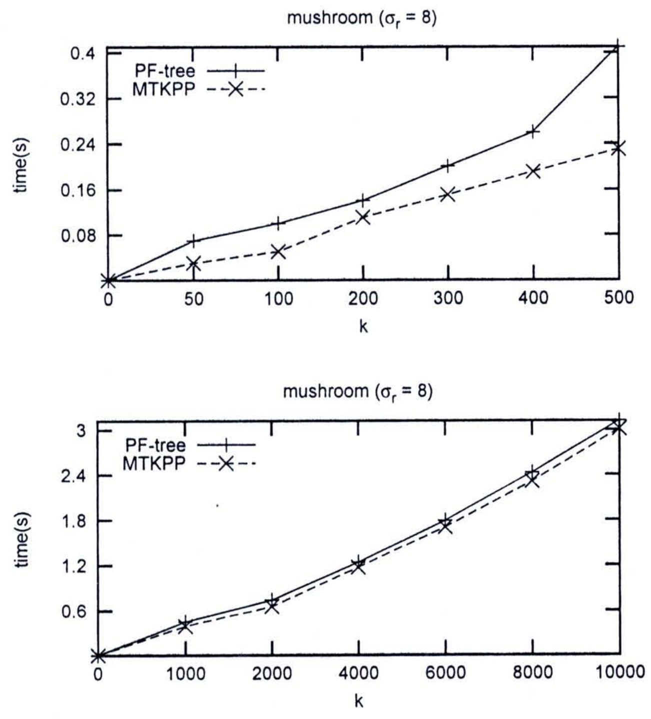
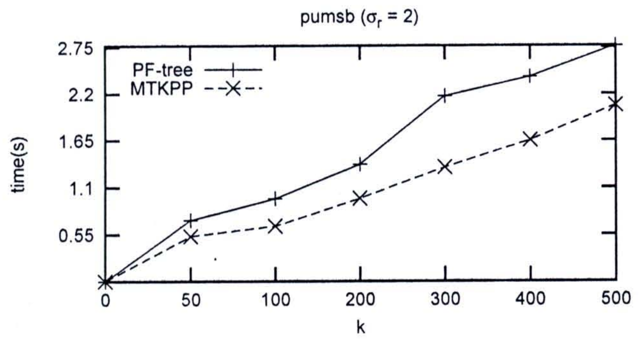


Figure 3.15: Runtime of MTKPP on *mushroom* ($\sigma_r = 8\%$)



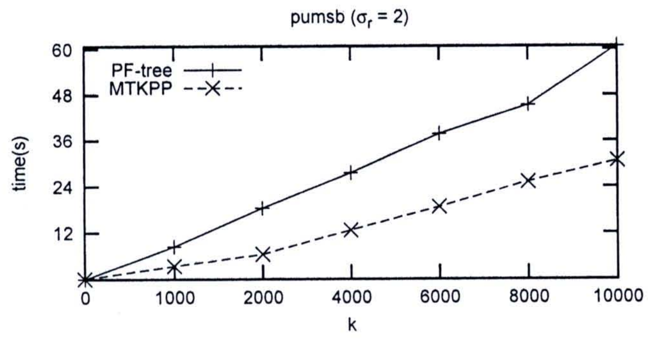
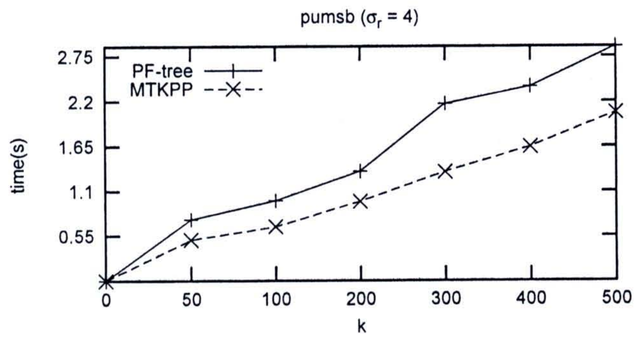


Figure 3.16: Runtime of MTKPP on *pumsb* ($\sigma_r = 2\%$)



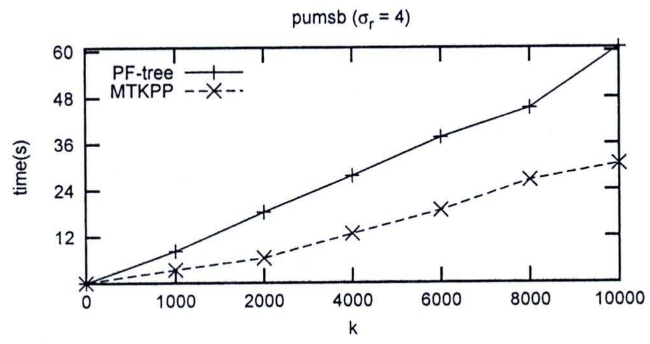
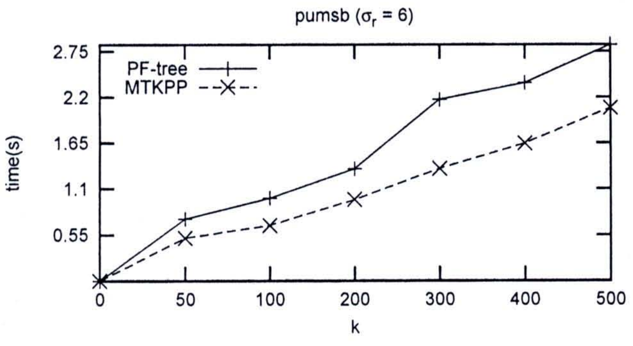


Figure 3.17: Runtime of MTKPP on *pumsb* ($\sigma_r = 4\%$)



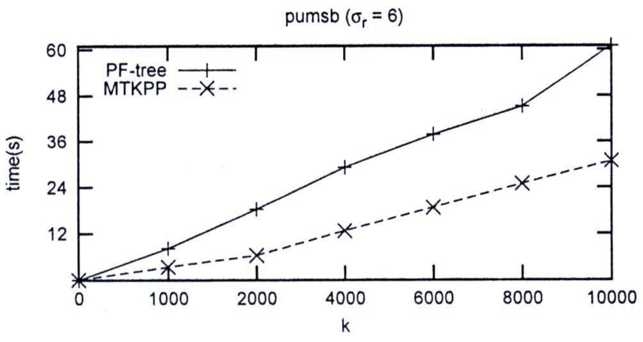
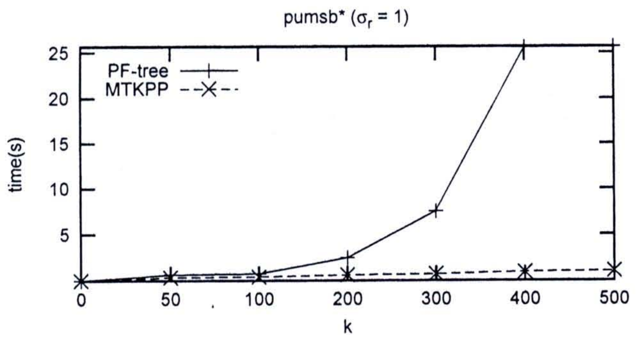


Figure 3.18: Runtime of MTKPP on *pumsb* ($\sigma_r = 6\%$)



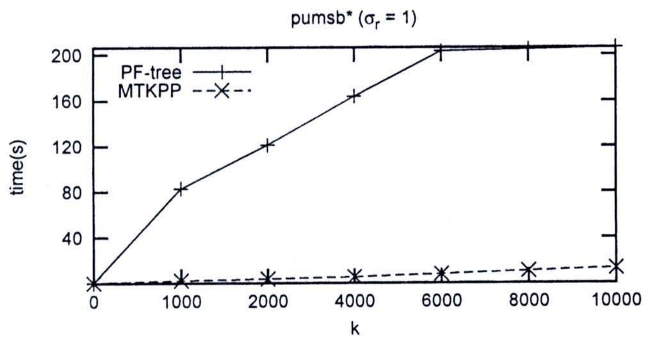
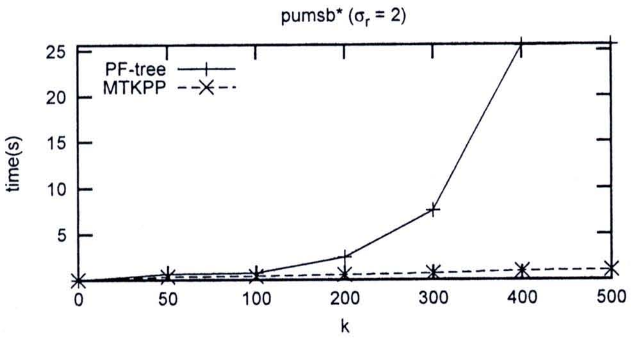


Figure 3.19: Runtime of MTKPP on *pumsb** ($\sigma_r = 1\%$)



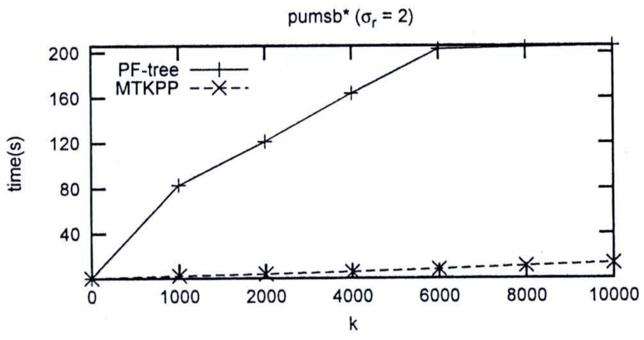
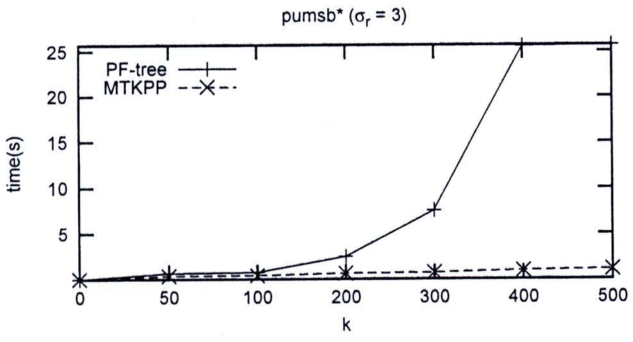


Figure 3.20: Runtime of MTKPP on *pumsb** ($\sigma_r = 2\%$)



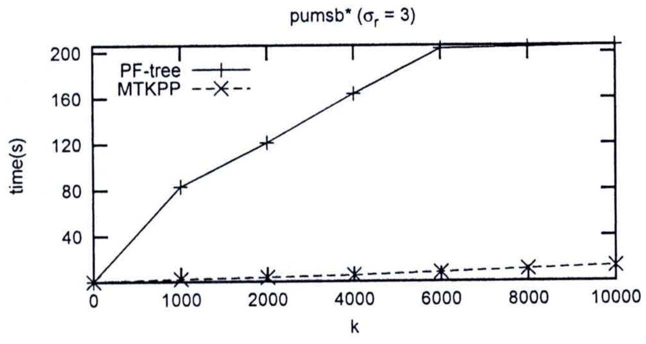
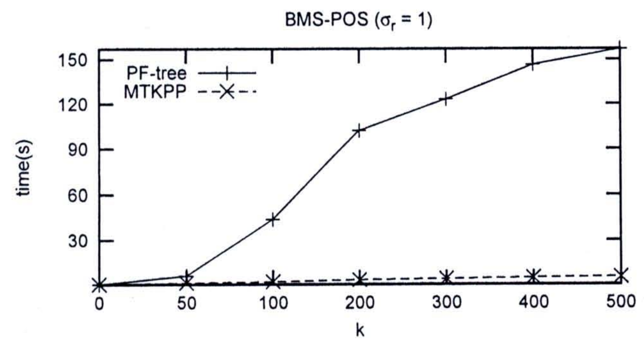


Figure 3.21: Runtime of MTKPP on *pumsb** ($\sigma_r = 3\%$)



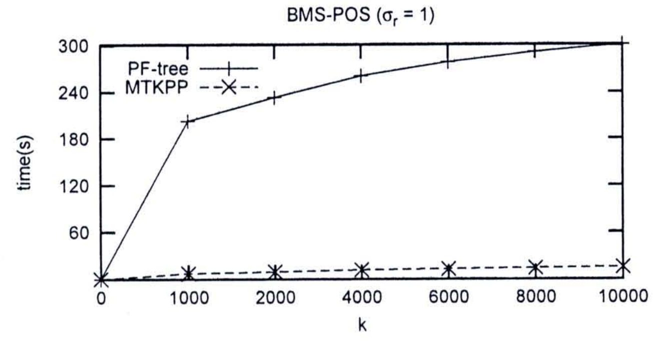
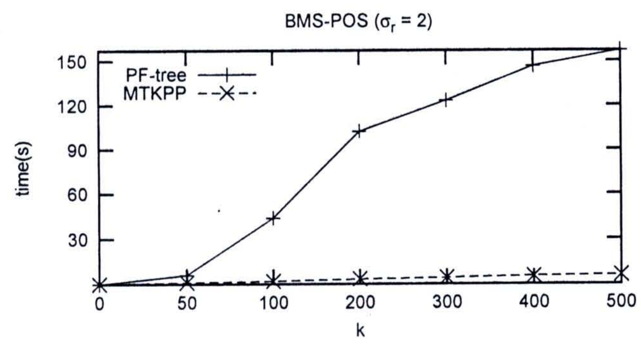


Figure 3.22: Runtime of MTKPP on *BMS-POS* ($\sigma_r = 1\%$)



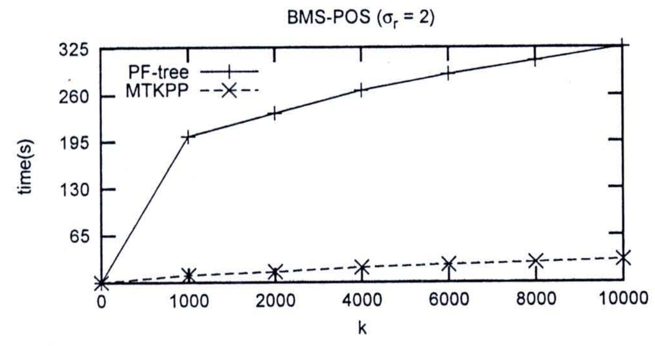


Figure 3.23: Runtime of MTKPP on *BMS-POS* ($\sigma_r = 2\%$)

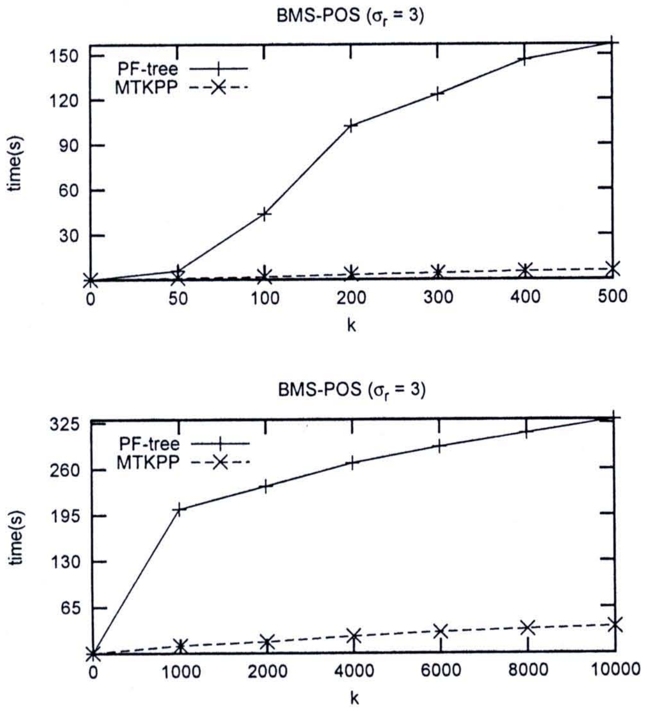


Figure 3.24: Runtime of MTKPP on *BMS-POS* ($\sigma_r = 3\%$)

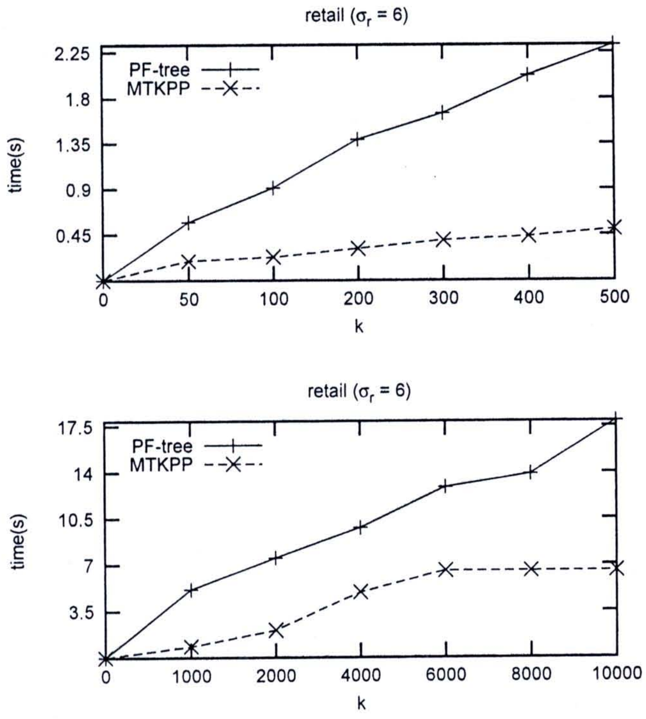


Figure 3.25: Runtime of MTKPP on *retail* ($\sigma_r = 6\%$)

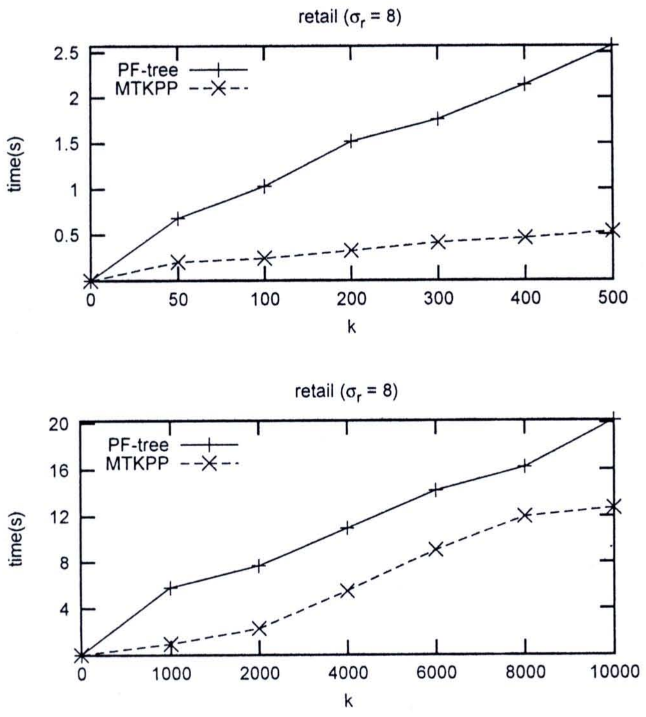


Figure 3.26: Runtime of MTKPP on *retail* ($\sigma_r = 8\%$)

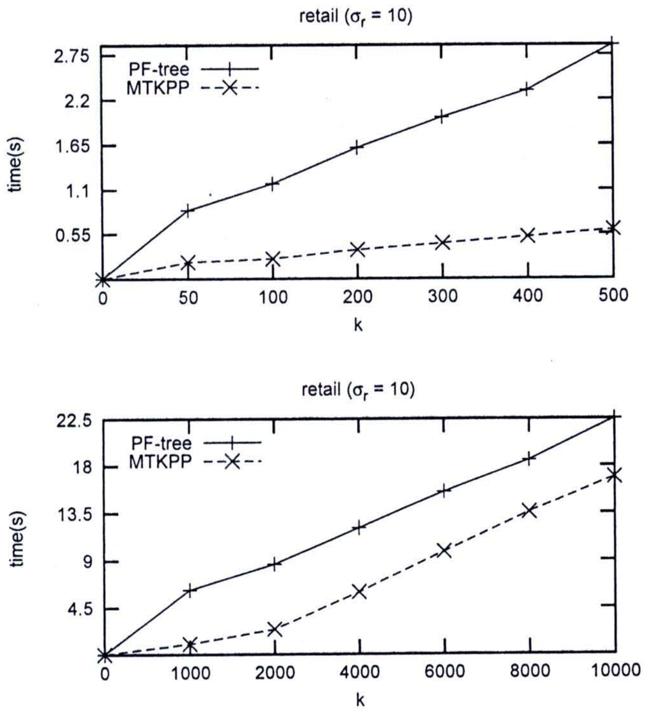


Figure 3.27: Runtime of MTKPP on *retail* ($\sigma_r = 10\%$)

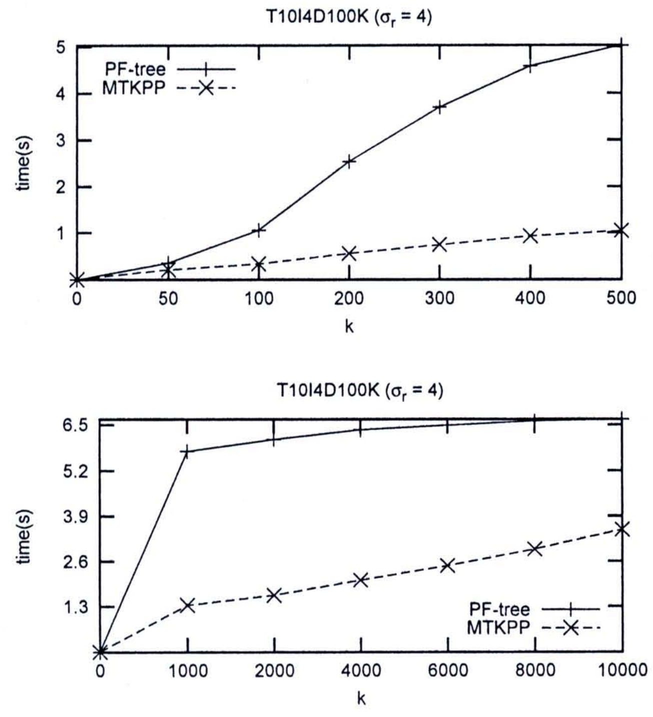


Figure 3.28: Runtime of MTKPP on *T10I4D100K* ($\sigma_r = 4\%$)

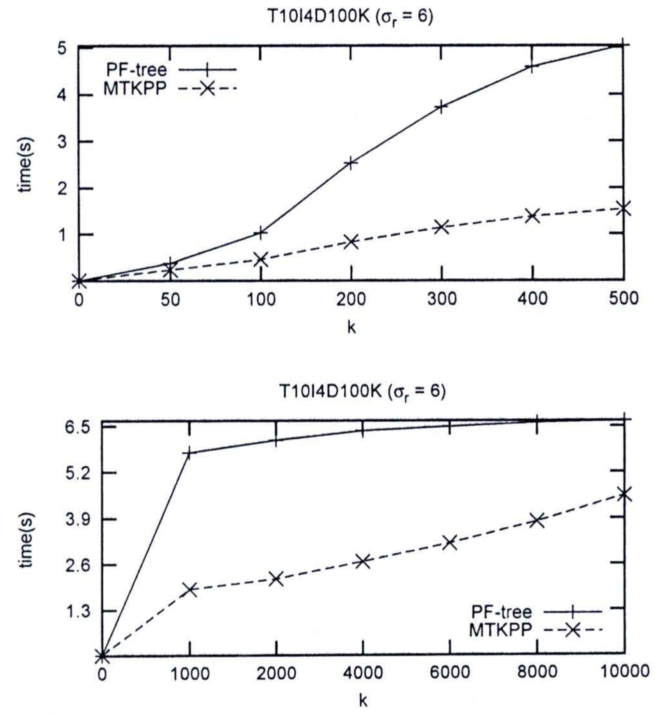


Figure 3.29: Runtime of MTKPP on *T10I4D100K* ($\sigma_r = 6\%$)

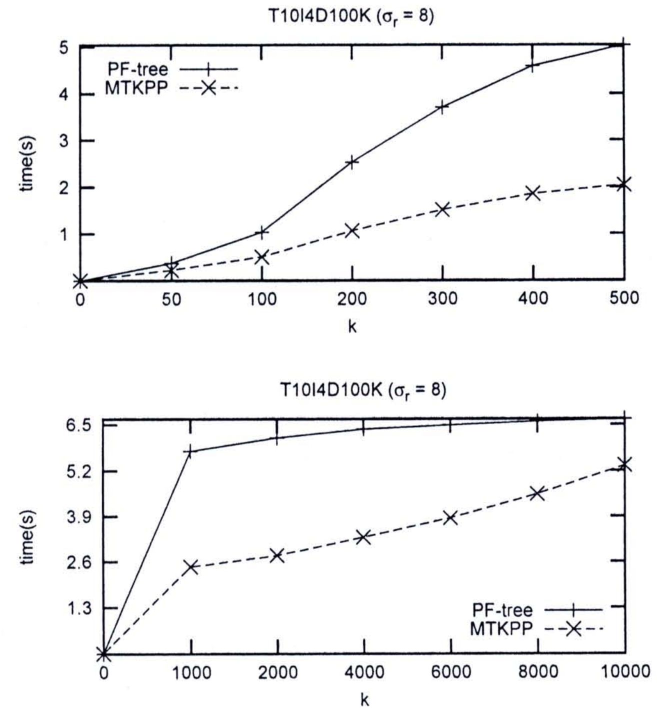


Figure 3.30: Runtime of MTKPP on *T10I4D100K* ($\sigma_r = 8\%$)

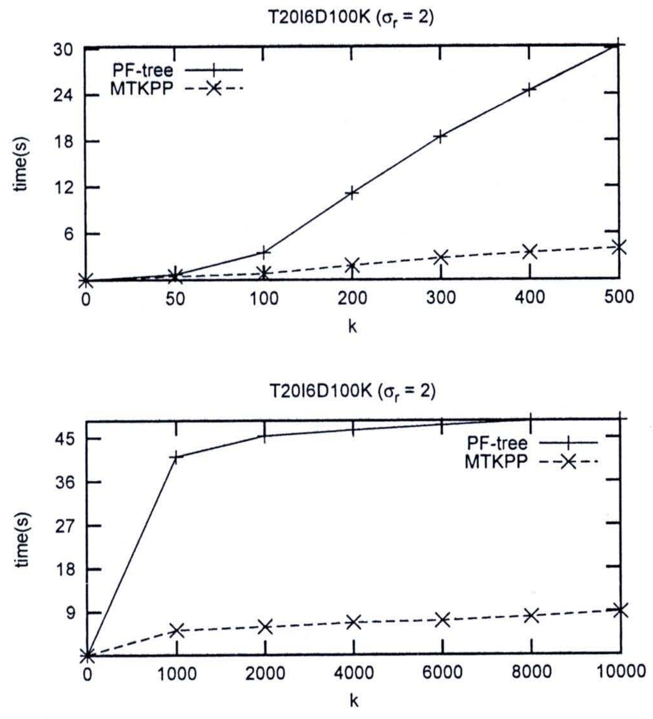
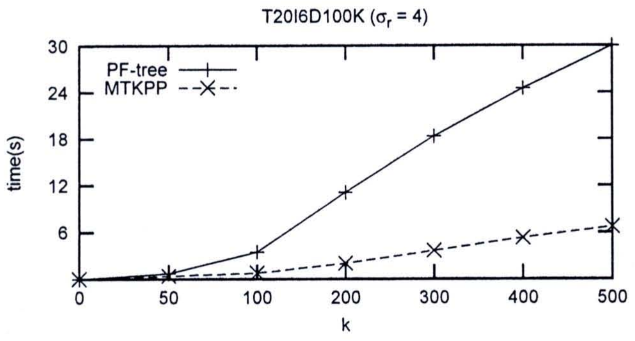


Figure 3.31: Runtime of MTKPP on *T20I6D100K* ($\sigma_r = 2\%$)



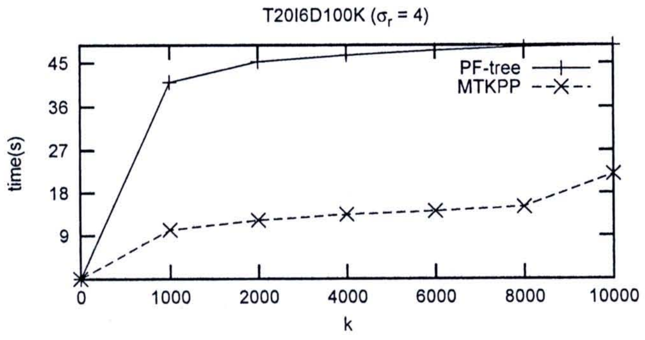
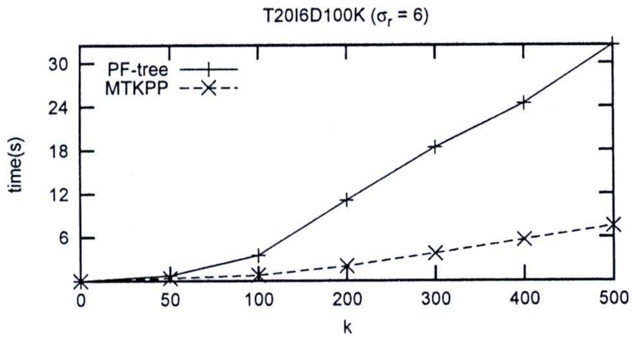


Figure 3.32: Runtime of MTKPP on *T20I6D100K* ($\sigma_r = 4\%$)



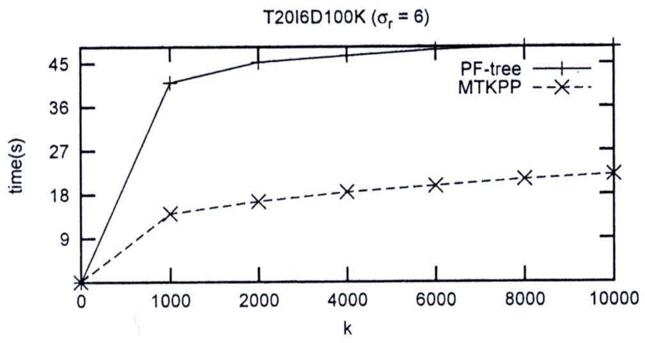
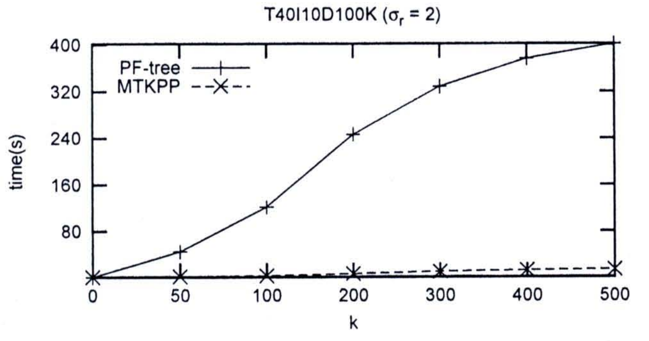


Figure 3.33: Runtime of MTKPP on *T20I6D100K* ($\sigma_r = 6\%$)



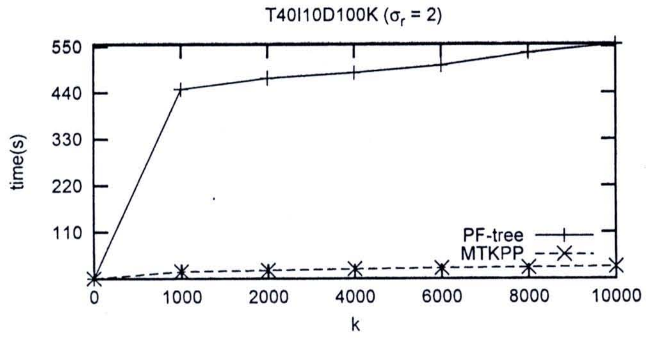
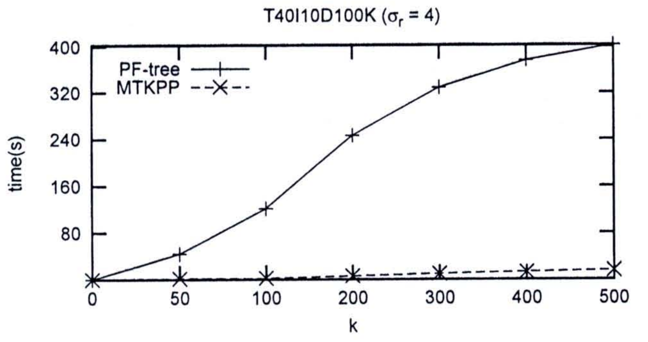


Figure 3.34: Runtime of MTKPP on *T40I10D100K* ($\sigma_r = 2\%$)



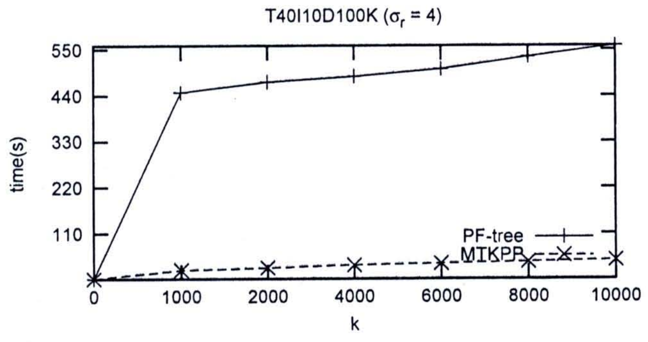


Figure 3.35: Runtime of MTKPP on *T40I10D100K* ($\sigma_r = 4\%$)

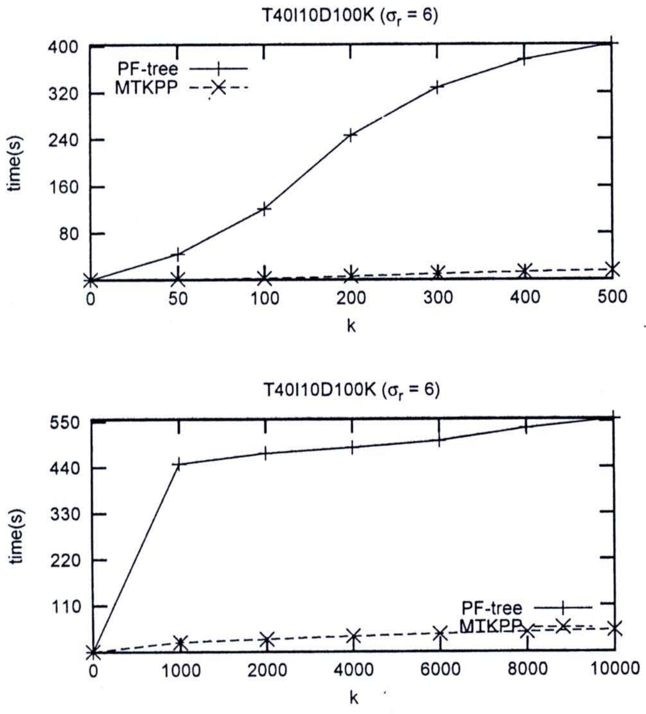


Figure 3.36: Runtime of MTKPP on *T40I10D100K* ($\sigma_r = 6\%$)

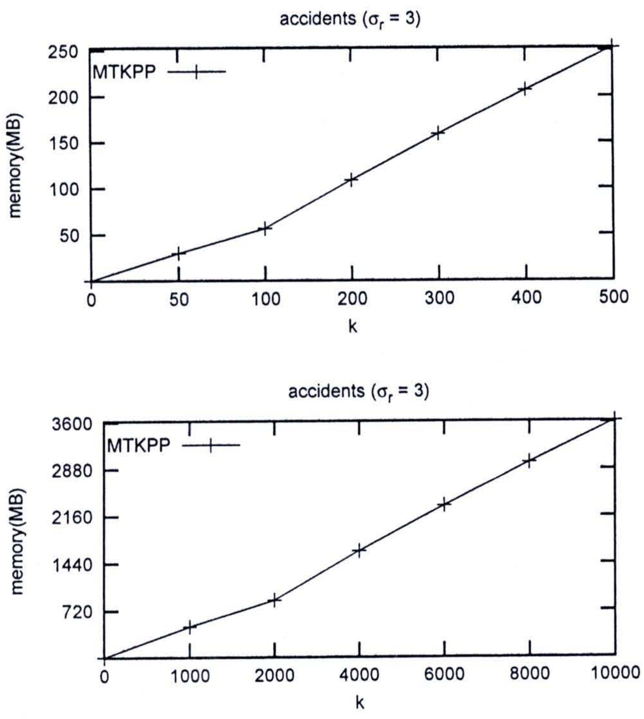


Figure 3.37: Memory usage of MTKPP on *accidents*

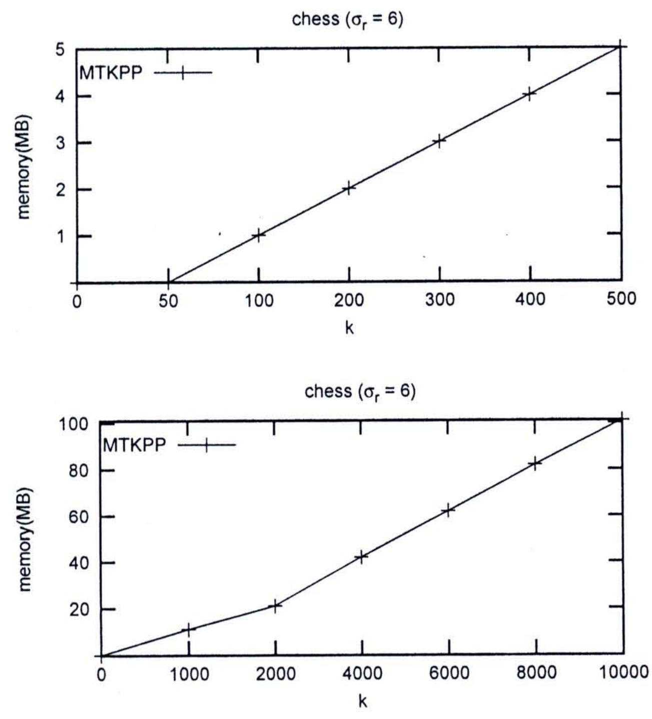


Figure 3.38: Memory usage of MTKPP on *chess*

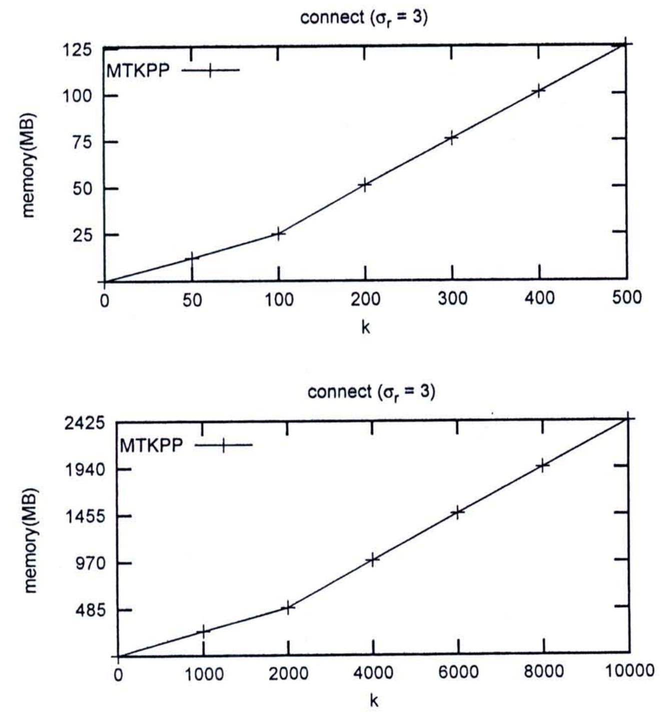


Figure 3.39: Memory usage of MTKPP on *connect*

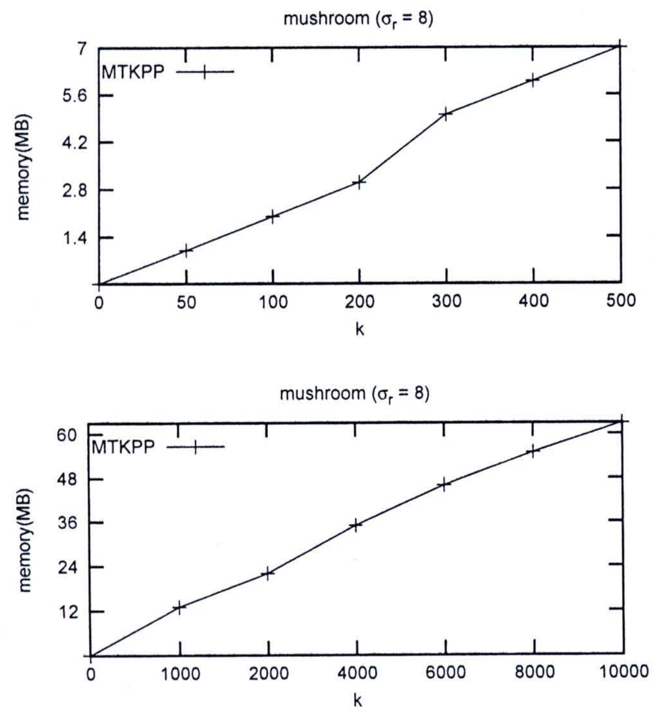


Figure 3.40: Memory usage of MTKPP on *mushroom*

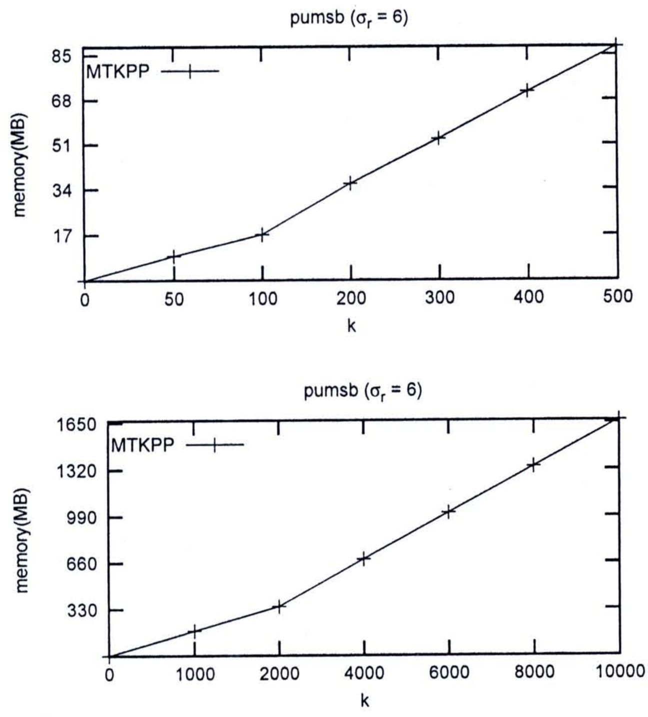
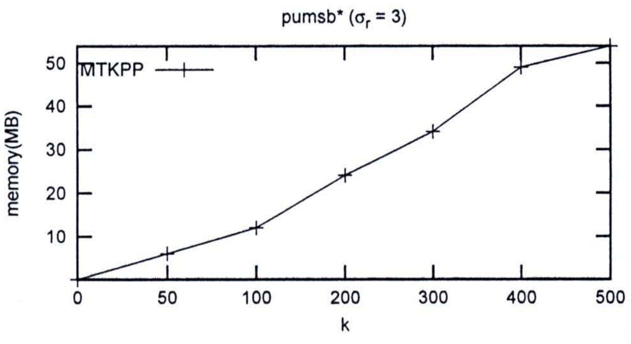


Figure 3.41: Memory usage of MTKPP on *pumsb*



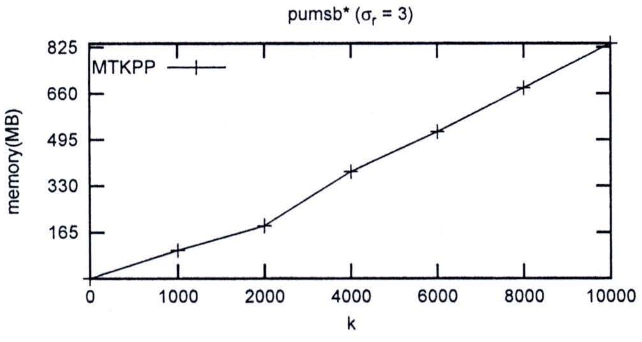
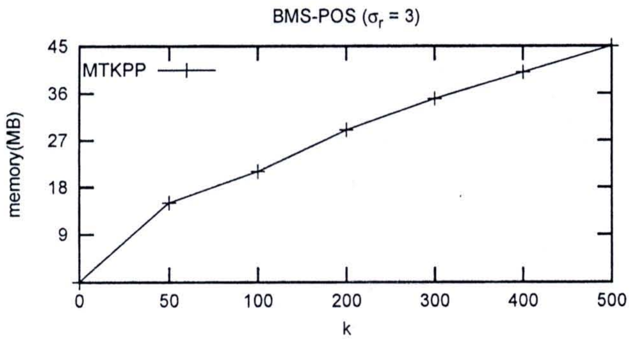


Figure 3.42: Memory usage of MTKPP on *pumsb**



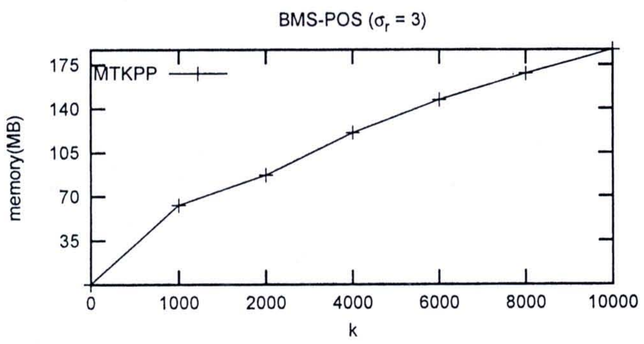
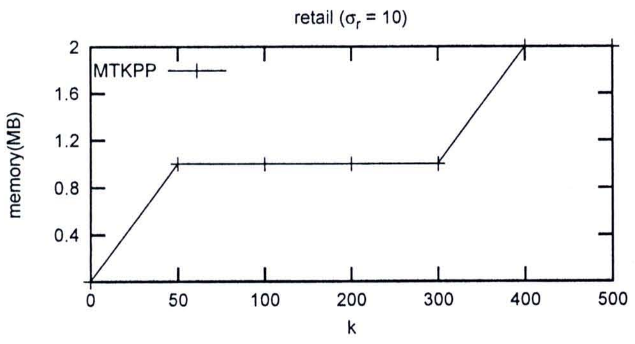


Figure 3.43: Memory usage of MTKPP on *BMS-POS*



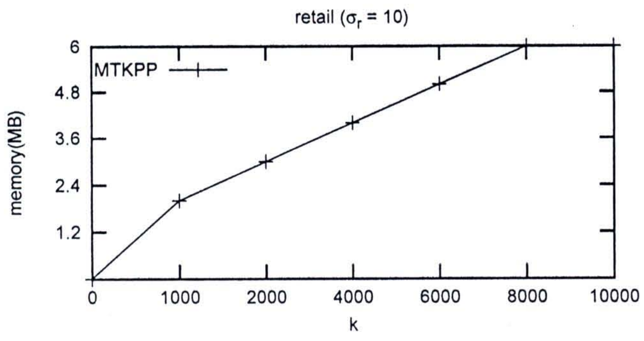
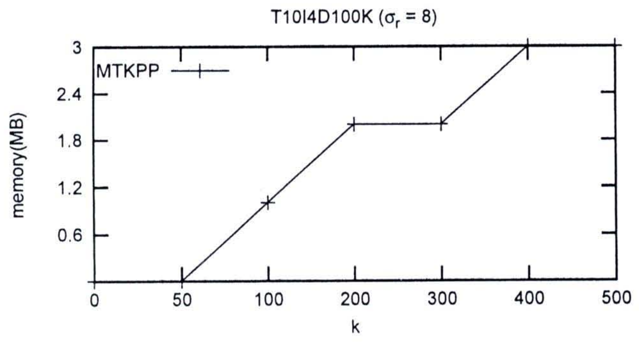


Figure 3.44: Memory usage of MTKPP on *retail*



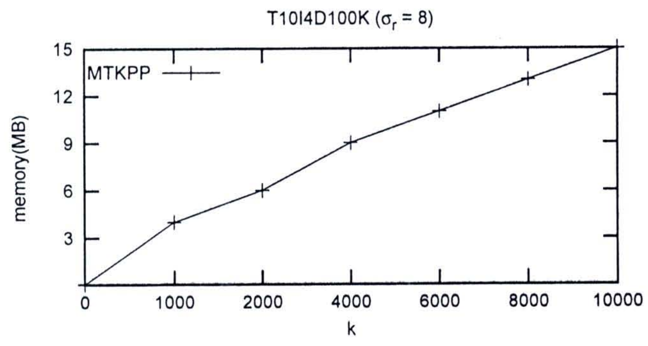
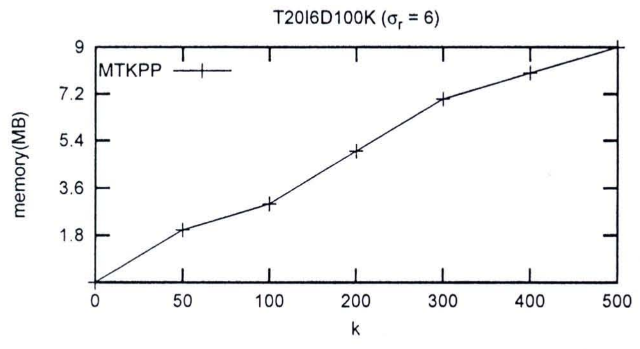


Figure 3.45: Memory usage of MTKPP on *T10I4D100K*



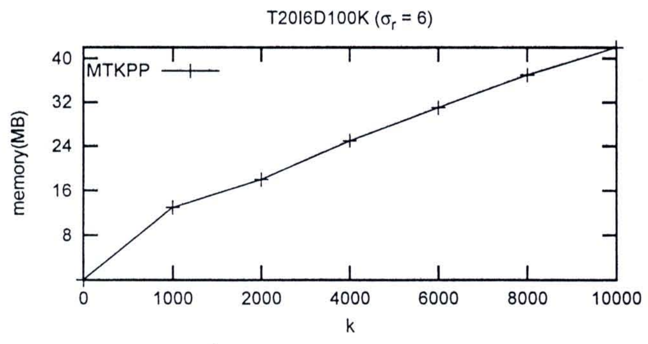
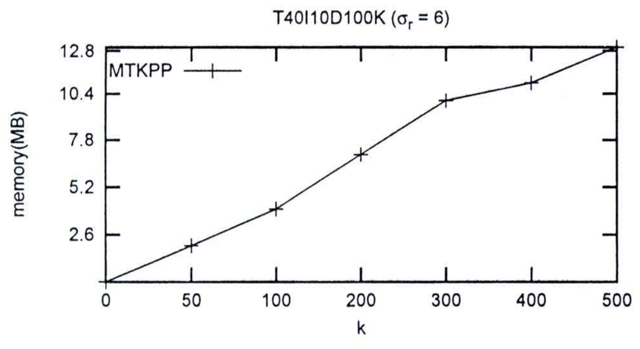


Figure 3.46: Memory usage of MTKPP on *T20I6D100K*



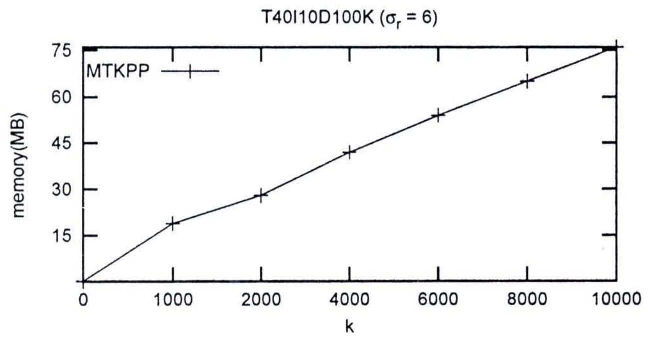
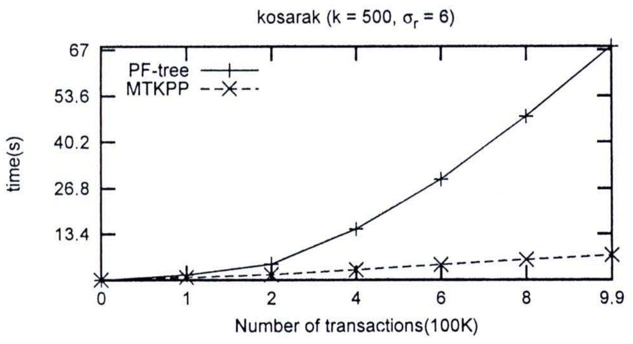


Figure 3.47: Memory usage of MTKPP on *T40I10D100K*



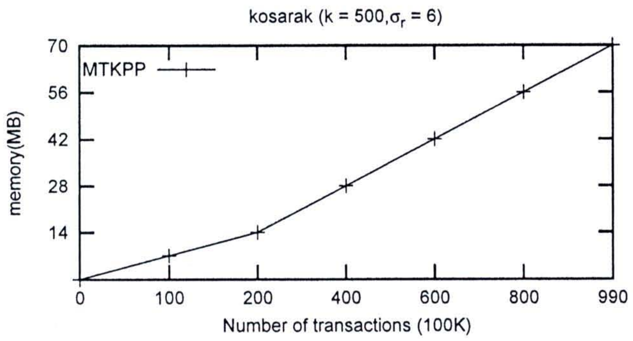
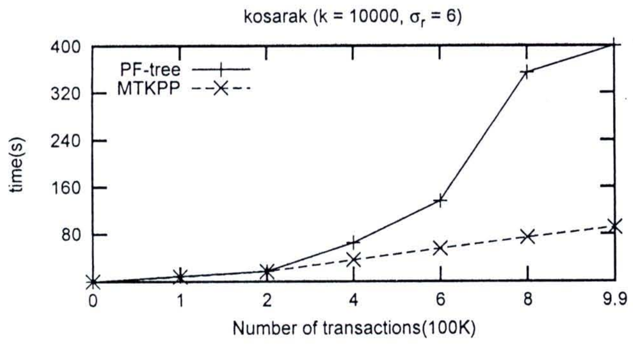


Figure 3.48: Scalability of MTKPP (k : $500, \sigma_r = 6$)



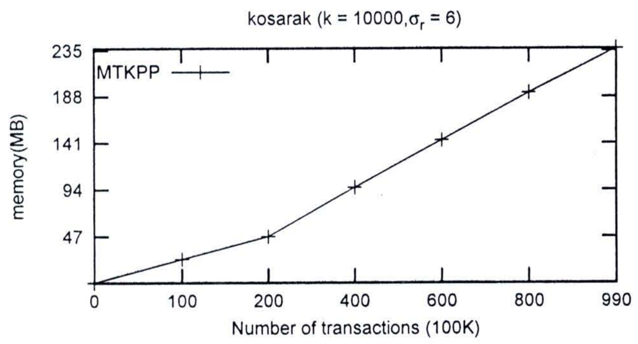


Figure 3.49: Scalability of MTKPP (k : $10,000, \sigma_r = 6$)