Project #1

My solution

Projection-based method

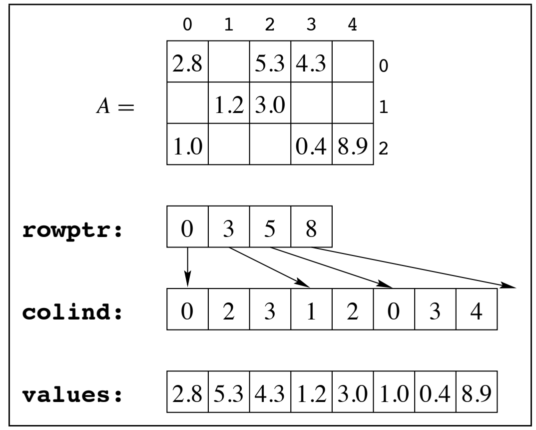
- Items are listed in lexicographic order
- Let P and DB(P) be a node's pattern and its associated projected database.
- Mining is performed by recursively calling this function:
  - TP(P, DB(P))
    1. Determine the frequent items in DB(P), and denote them by E(P).
    2. Eliminate from DB(P) any items not in E(P).
    3. For each item x in E(P), call TP(Px, DB(Px)).

What is the right data structure for storing the projected database?

- Operations that needs to support:
  - Determining the frequency of each item.
  - Projection:
    - Extract the transactions and their subsets of items that support a given pattern.
- What should be the upper bound on the complexity of each of these operations?

Dense matrices are stored in the computer memory by using two-dimensional arrays. For example, a matrix with n rows and m columns, is stored using a $n \times m$ array of real numbers. However, using the same two-dimensional arrays to store sparse matrices has two very important drawbacks. First, since most of the entries in the sparse matrix are zero, this storage scheme wastes a lot of memory. Second, computations involving sparse matrices often need to operate only on the non-zero entries of the matrix. Use of dense storage format makes it harder to locate these non-zero entries. For these reasons sparse matrices are stored using different data structures.

The compressed storage format (CSR) is a widely used scheme for storing sparse matrices. In the CSR format, an $n \times n$ sparse matrix A that has k non-zero entries is stored using three arrays: two integer arrays `rowptr` and `colind`, and one array of real values `values`. The array `rowptr` is of size $n+1$, and the other two arrays are each of size k. The array `colind` stores the column-indices of the non-zero entries in A, and the array `values` stores the corresponding non-zero entries. In particular, the array `colind` stores the column-indices of the first row followed by the column-indices of the second row followed by the column-indices of the third row, and so on. Similarly, the array `values` stores the corresponding non-zero entries of the first row followed by the corresponding non-zero entries of the second row, and so on. The array `rowptr` is used to determine where the storage of a row starts and ends in the arrays `colind` and `values`. In particular, the column-indices of row i are stored starting at `colind[rowptr[i]]` and ending at (but not including) `colind[rowptr[i+1]]`. Similarly, the values of the non-zero entries of row i are stored starting at `values[rowptr[i]]` and ending at (but not including) `values[rowptr[i+1]]`. Also note that the number of non-zero entries for row i is simply `rowptr[i+1]-rowptr[i]`.



CSR format of a sample matrix

Figure 0.11: The compressed storage format for sparse matrices.

My naming of the CSR/CSC fields is different from the common way of doing things.
I use "row" as the prefix for the row-wise representation (`rowptr, rowind, rowval`).
I use "col" as the prefix for the column-wise representation (`colptr, colind, colval`).

```
/*-------------------------------------------------------------
 * The following data structure stores a sparse CSR format
 *-------------------------------------------------------------*/
typedef struct gk_csr_t {
  int32_t nrows, ncols;
  ssize_t *rowptr, *colptr;
  int32_t *rowind, *colind;
  int32_t *rowids, *colids;
  int32_t *rlabels, *clabels;
  int32_t *rmap, *cmap;
  float *rowval, *colval;
  float *rnorms, *cnorms;
  float *rsums, *csums;
  float *rsizes, *csizes;
  float *rvols, *cvols;
  float *rwgts, *cwgts;
} gk_csr_t;
```

```c
/*-------------------------------------------------------------*/
/*! Data structures for use within this module */
/*-------------------------------------------------------------*/
typedef struct {
  int minfreq;  /* the minimum frequency of a pattern */
  int maxfreq;  /* the maximum frequency of a pattern */
  int minlen;   /* the minimum length of the requested pattern */
  int maxlen;   /* the maximum length of the requested pattern */
  int tnitems;  /* the initial range of the item space */

  /* the call-back function */
  void (*callback)(void *stateptr, int nitems, int *itemids, int ntrans, int *transids);
  void *stateptr;   /* the user-supplied pointer to pass to the callback */

  /* workspace variables */
  int *rmarker;
  gk_ikv_t *cand;
} isparams_t;
```

The memory for those is allocated once and reused.

This is to keep track of the original item-IDs, since their numbering will change due to reordering.

```c
/*************************************************************************/
/*! The entry point of the frequent itemset discovery code */
/*************************************************************************/
void gk_find_frequent_itemsets(int ntrans, ssize_t *tranptr, int *tranind,
        int minfreq, int maxfreq, int minlen, int maxlen,
        void (*process_itemset)(void *stateptr, int nitems, int *itemids,
                                int ntrans, int *transids),
        void *stateptr)
{
  ssize_t i;
  gk_csr_t *mat, *pmat;
  isparams_t params;
  int *pattern;

  /* Create the matrix */
  mat = gk_csr_Create();
  mat->nrows  = ntrans;
  mat->ncols  = tranind[gk_iargmax(tranptr[ntrans], tranind, 1)]+1;
  mat->rowptr = gk_zcopy(ntrans+1, tranptr,
                  gk_zmalloc(ntrans+1, "gk_find_frequent_itemsets: mat.rowptr"));
  mat->rowind = gk_icopy(tranptr[ntrans], tranind,
                  gk_imalloc(tranptr[ntrans], "gk_find_frequent_itemsets: mat.rowind"));
  mat->colids = gk_iincset(mat->ncols, 0,
                  gk_imalloc(mat->ncols, "gk_find_frequent_itemsets: mat.colids"));

  /* Setup the parameters */
  params.minfreq  = minfreq;
  params.maxfreq  = (maxfreq == -1 ? mat->nrows : maxfreq);
  params.minlen   = minlen;
  params.maxlen   = (maxlen == -1 ? mat->ncols : maxlen);
  params.tnitems  = mat->ncols;
  params.callback = process_itemset;
  params.stateptr = stateptr;
  params.rmarker  = gk_ismalloc(mat->nrows, 0, "gk_find_frequent_itemsets: rmarker");
  params.cand     = gk_ikvmalloc(mat->ncols, "gk_find_frequent_itemsets: cand");

  /* Perform the initial projection */
  gk_csr_CreateIndex(mat, GK_CSR_COL);
  pmat = itemsets_project_matrix(&params, mat, -1);
  gk_csr_Free(&mat);

  pattern = gk_imalloc(pmat->ncols, "gk_find_frequent_itemsets: pattern");
  itemsets_find_frequent_itemsets(&params, pmat, 0, pattern);

  gk_csr_Free(&pmat);
  gk_free((void **)&pattern, &params.rmarker, &params.cand, LTERM);

}
```

```c
/*************************************************************************/
/*! The recursive routine for DFS-based frequent pattern discovery */
/*************************************************************************/
void itemsets_find_frequent_itemsets(isparams_t *params, gk_csr_t *mat,
         int preflen, int *prefix)
{
  ssize_t i;
  gk_csr_t *cmat;

  /* Project each frequent column */
  for (i=0; i<mat->ncols; i++) {
    prefix[preflen] = mat->colids[i];

    if (preflen+1 >= params->minlen)
      (*params->callback)(params->stateptr, preflen+1, prefix,
           mat->colptr[i+1]-mat->colptr[i], mat->colind+mat->colptr[i]);

    if (preflen+1 < params->maxlen) {
      cmat = itemsets_project_matrix(params, mat, i);
      itemsets_find_frequent_itemsets(params, cmat, preflen+1, prefix);
      gk_csr_Free(&cmat);
    }
  }

}
```

Why?

```c
/*****************************************************************************/
/*! This function projects a matrix w.r.t. to a particular column.
    It performs the following steps:
    - Determines the length of each column that is remaining.
    - Sorts the columns in increasing length.
    - Creates a column-based version of the matrix with the proper
      column ordering.
 */
/*****************************************************************************/
gk_csr_t *itemsets_project_matrix(isparams_t *params, gk_csr_t *mat, int cid)
```

```c
gk_csr_t *itemsets_project_matrix(isparams_t *params, gk_csr_t *mat, int cid)
{
  ssize_t i, j, k, ii, pnnz;
  int nrows, ncols, pnrows, pncols;
  ssize_t *colptr, *pcolptr;
  int *colind, *colids, *pcolind, *pcolids, *rmarker;
  gk_csr_t *pmat;
  gk_ikv_t *cand;

  nrows  = mat->nrows;
  ncols  = mat->ncols;
  colptr = mat->colptr;
  colind = mat->colind;
  colids = mat->colids;

  rmarker = params->rmarker;
  cand    = params->cand;

  /* Allocate space for the projected matrix based on what you know thus far */
  pmat = gk_csr_Create();
  pmat->nrows  = pnrows = (cid == -1 ? nrows : colptr[cid+1]-colptr[cid]);

  /* Mark the rows that will be kept and determine the prowids */
  if (cid == -1) { /* Initial projection */
    gk_iset(nrows, 1, rmarker);
  }
  else { /* The other projections */
    for (i=colptr[cid]; i<colptr[cid+1]; i++)
      rmarker[colind[i]] = 1;
  }

  /* Determine the length of each column that will be left in the projected matrix */
  for (pncols=0, pnnz=0, i=cid+1; i<ncols; i++) {
    for (k=0, j=colptr[i]; j<colptr[i+1]; j++) {
      k += rmarker[colind[j]];
    }
    if (k >= params->minfreq && k <= params->maxfreq) {
      cand[pncols].val   = i;
      cand[pncols++].key = k;
      pnnz += k;
    }
  }

  /* Sort the columns in increasing order */
  gk_ikvsorti(pncols, cand);
```

This is either 0 or 1.

pncols: is the number of columns in the projected database.

pnnz: is the total number of items in the transactions of the projected database.

```c
/* Sort the columns in increasing order */
gk_ikvsorti(pncols, cand);

/* Allocate space for the remaining fields of the projected matrix */
pmat->ncols  = pncols;
pmat->colids = pcolids = gk_imalloc(pncols, "itemsets_project_matrix: pcolids");
pmat->colptr = pcolptr = gk_zmalloc(pncols+1, "itemsets_project_matrix: pcolptr");
pmat->colind = pcolind = gk_imalloc(pnnz, "itemsets_project_matrix: pcolind");

/* Populate the projected matrix */
pcolptr[0] = 0;
for (pnnz=0, ii=0; ii<pncols; ii++) {
  i = cand[ii].val;
  for (j=colptr[i]; j<colptr[i+1]; j++) {
    if (rmarker[colind[j]])
      pcolind[pnnz++] = colind[j];
  }

  pcolids[ii] = colids[i];
  pcolptr[ii+1] = pnnz;
}

/* Reset the rmarker array */
if (cid == -1) { /* Initial projection */
  gk_iset(nrows, 0, rmarker);
}
else { /* The other projections */
  for (i=colptr[cid]; i<colptr[cid+1]; i++)
    rmarker[colind[i]] = 0;
}

return pmat;
}
```

This optimization is essential for sparse data.