KMC API documentation

for KMC v. 2.3.0

Contents

Introduction

This document describe how to access k-mers stored in database created by KMC k-mer counter. Section 1 contains a description of class provided to work with KMC database. Section 2 contains an example of API usage to store all k-mers with counters in text file. Section 3 contains a detailed description how databse is build. This description refers to database created with KMC 2.x (previous version of database format is described in supplementary material of KMC 1 paper - http://www.biomedcentral.com/content/supplementary/1471-2105-14-160-S1.pdf). From version 2.2.0 API can handle both database formats. Any new feature/bug fix for API is added only for KMC2.x branch, standalone API for previous versions is not longer under development, so new version of API should be used even for databases produced by older KMC versions.

1 API

In this section we describe two classes, CKmerAPI and CKMCFile. They can be used to obtain access to the databases produced by KMC program.

1.1 CKmerAPI class

This class represents a k-mer. Its key methods are:

- CKmerAPI(uint32 length = 0) — constructor, that creates the array kmer_data of appropriate size,
- CKmerAPI(const CKmerAPI &kmer) — copy constructor,
- char get_asci_symbol(unsigned int pos) — returns k-mer's symbol at a given position (0-based), the symbol is in ASCII representation (ACGT)
- uchar get_num_symbol(unsigned int pos) — returns k-mer's symbol at a given position (0-based), the symbol is in numberic representation (0, 1, 2, 3)
- std::string to_string() — converts k-mer to string, using the alphabet ACGT,
- void to_string(char *str) — converts k-mer to string, using the alphabet ACGT; the function assumes that enough memory was allocated,
- void to_string(str::string &str) — converts k-mer to string, using the alphabet ACGT,
- bool from_string(const std::string &str) — converts string (from alphabet ACGT) to k-mer,
- bool from_string(const char *kmer_string) — converts string (from alphabet ACGT) to k-mer,
- bool reverse() — Convert k-mer to its reverse complement
- $\sim$CKmerAPI() — destructor, releases the content of kmer_data array,
- overloaded operators: =, ==, $<$.

1.2 CKMCFile class

This class handles a k-mer database. Its key methods are:

- CKMCFile() — constructor,
- bool OpenForRA(const std::string &file_name) — opens two files: file_name with added extension ".kmc_pre" and ".kmc_suf", reads their whole content to enable random access (in memory), and then closes them,
- bool OpenForListing(const std::string &file_name) — opens the file file_name with added extension ".kmc_pre" and allows to read the k-mers one by one (whole database is not loaded into memory),
- bool ReadNextKmer(CKmerAPI &kmer, uint32 &count) — reads next k-mer to kmer and updates its count; the return value is bool; true as long as not eof-of-file (available only when database is opened in listing mode), this method should **not** be used for Quake compatibile mode, but for direct counters; for Quake compatibile counters use overload which takes counter as float&,

- bool ReadNextKmer(CKmerAPI &kmer, float &count) — equivalent of ReadNextKmer(CKmerAPI &kmer, uint32 &count) for Quake compatibile mode,
- bool ReadNextKmer(CKmerAPI &kmer, uint64 &count) — equivalent of ReadNextKmer(CKmerAPI &kmer, uint32 &count) with counter as uint64; for small k values (i.e. $k < 10$) it is possible to run KMC to store counters as uint64,
- bool Close() — if the file was opened for random access, the allocated memory for its content is released; if the file was opened for listing, the allocated memory for its content is released and the ".kmer" file is closed,
- bool SetMinCount(uint32 x) — set the minimum counter value for k-mers; if a k-mer has count below x, it is treated as non-existent,
- uint32 GetMinCount(void) — returns the value (uint32) set with SetMinCount,
- bool SetMaxCount(uint32 x) — set the maximum counter value for k-mers; if a k-mer has count above x, it is treated as non-existent,
- uint64 GetMaxCount(void) — returns the value (uint32) set with SetMaxCount,
- uint64 KmerCount(void) — returns the number of k-mers in the database (available only for databases opened in random access mode),
- bool GetBothStrands(void) — returns true if KMC was launched without -b swich (i.e. k-mers was transformed to canonical form),
- uint32 KmerLength(void) — returns the k-mer length in the database (available only for databases opened in random access mode),
- bool RestartListing(void) — sets the cursor for listing k-mers from the beginning of the file (available only for databases opened in listing mode). The method OpenForListing(std::string file_name) invokes it automatically, but it can be also called by a user,
- bool Eof(void) — returns true if all k-mers have been listed,
- bool CheckKmer(CKmerAPI &kmer, uint32 &count) — returns true if kmer exists in the database and set its count if the answer is positive (available only for databases opened in random access mode), this method should **not** be used for Quake compatibile mode, but for direct counters; for Quake compatibile counters use overload which takes counter as float&,
- bool CheckKmer(CKmerAPI &kmer, float &count) — equivalent of CheckKmertKmer(CKmerAPI &kmer, uint32 &count) for Quake compatibile mode,
- bool CheckKmer(CKmerAPI &kmer, uint64 &count) — equivalent of bool CheckKmer(CKmerAPI &kmer, uint32 &count) with counter as uint64; for small k values (i.e. $k < 10$) it is possible to run KMC to store counters as uint64,
- bool IsKmer(CKmerAPI &kmer) — returns true if kmer exists (available only for databases opened in random access mode),
- void ResetMinMaxCounts(void) — sets min_count and max_count to the values read from the database,

- bool Info(uint32 &_kmer_length, uint32 &_mode, uint32 &_counter_size, uint32 &_lut_prefix_length, uint32 &_signature_len, uint32 &_min_count, uint32 &_max_count, uint64 &_total_kmers) — gets current parameters from the k-mer database (if the database is in KMC 1 format the signature_len is set to 0); from ver. 2.3.0 it is recommended to use bool Info(CKMCFileInfo& info);,
- bool Info(CKMCFileInfo& info) — gets current parameters from the k-mer database as a CKMCFileInfo object; CKMCFileInfo class contains: uint32 kmer_length, uint32 mode, uint32 counter_size, uint32 lut_prefix_length, uint32 signature_len, uint32 min_count, uint64 max_count bool both_strands, uint64 total_kmers,
- bool GetCountersForRead(const std::string& read, std::vector<uint32>& counters) — splits input *read* into k-mers and for each set its counter into output parameter *counters*; if some k-mer is invalid or does not exists in database its counter is set to 0; the size of *counters* is always equal to *read.lenght() - k + 1)* (available only for databases opened in random access mode), this method should **not** be used for Quake compatibile mode, but for direct counters; for Quake compatibile counters use overload which takes *counters* as std::vector<float>& counters,
- bool GetCountersForRead(const std::string& read, std::vector<float>& counters) — equivalent of GetCountersForRead(const std::string& read, std::vector<uint32>& counters) for Quake compatibile mode
- $\sim$CKMCFile() — destructor.

2 Example of API usage

The kmc_dump application (Figs. 1 and 2) shows how to list and print k-mers with at least min_count and at most max_count occurrences in the database. Fig. 1 presents parsing the command-line parameters, including -ci<value> and -cx<value>. Input and output file names are also expected. The code in Fig. 2 is for actual database handling. This database is represented by a CKMCFile object, which opens an input file for k-mer listing (the method bool OpenForListing(std::string file_name) is invoked). The parameter of the method SetMinCount (SetMaxCount) must be not smaller (not greater) than the corresponding parameter -ci (-cx) with which KMC was invoked (otherwise, nothing will be listed). The listed k-mers are in the form like:
AAACACCGT\t<value>
where the first part is the k-mer in natural representation, which is followed by a tab character, and its associated value (integer or float). (Such format is compatible with Quake, a widely used tool for sequencing error correction.) Note that, if needed, one can easily modify the output format, changing the lines 32 and 41 in Fig. 2.

For performance reasons, the KMC package contains two variants of the dump program. The first one, presented below, is the kmc_dump_sample program. The second variant, kmc_dump, is essentially the same, the only difference is the way the counters are printed. Instead of the fprintf function we used much faster way of converting numbers into the textual form. Thus, in real applications the kmc_dump variant should be used.

```
#include <iostream>
#include "../kmc_api/kmc_file.h"

void print_info(void);

int _tmain(int argc, char* argv[])
{
  CKMCFile kmer_database;
  int i;
  uint32 min_count_to_set = 0;
  uint32 max_count_to_set = 0;
  std::string input_file_name;
  std::string output_file_name;

  FILE * out_file;
  //-----------------------------------------------------------
  // Parse input parameters
  //-----------------------------------------------------------
  if(argc < 3)
  {
    print_info();
    return EXIT_FAILURE;
  }

  for(i = 1; i < argc; ++i)
  {
    if(argv[i][0] == '-')
    {
      if(strncmp(argv[i], "-ci", 3) == 0)
        min_count_to_set = atoi(&argv[i][3]);
      else if(strncmp(argv[i], "-cx", 3) == 0)
          max_count_to_set = atoi(&argv[i][3]);
    }
    else
      break;
  }

  if(argc - i < 2)
  {
    print_info();
    return EXIT_FAILURE;
  }

  input_file_name = std::string(argv[i++]);
  output_file_name = std::string(argv[i]);

  if((out_file = fopen (output_file_name.c_str(),"wb")) == NULL)
  {
    print_info();
    return EXIT_FAILURE;
  }

  setvbuf(out_file, NULL ,_IOFBF, 1 << 24);

  ...
```

Figure 1: First part of kmc_dump_sample application

```
    //------------------------------------------------------------------------------
    // Open kmer database for listing and print kmers within min_count and max_count
    //------------------------------------------------------------------------------

    if(!kmer_database.OpenForListing(input_file_name))
    {
      print_info();
      return EXIT_FAILURE ;
    }
    else
    {
      uint32 _kmer_length, _mode,_counter_size, _lut_prefix_length, _signature_len, _min_count,  _max_count;
      uint64 _total_kmers;

      kmer_data_base.Info(_kmer_length, _mode, _counter_size, _lut_prefix_length, _signature_len, _min_count, _max_cour

      CKmerAPI kmer_object(_kmer_length);

      if(min_count_to_set)
        if (!(kmer_data_base.SetMinCount(min_count_to_set)))
          return EXIT_FAILURE;
      if(max_count_to_set)
        if (!(kmer_data_base.SetMaxCount(max_count_to_set)))
          return EXIT_FAILURE;
      std::string str;
      if (_mode) //quake compatible mode
      {
        float counter;
        while (kmer_data_base.ReadNextKmer(kmer_object, counter))
        {
          kmer_object.to_string(str);
          fprintf(out_file, "%s\t%f\n", str.c_str(), counter);
        }
      }
      else
      {
        uint32 counter;
        while (kmer_data_base.ReadNextKmer(kmer_object, counter))
        {
          kmer_object.to_string(str);
          fprintf(out_file, "%s\t%u\n", str.c_str(), counter);
        }
      }


      fclose(out_file);
      kmer_data_base.Close();
    }

    return EXIT_SUCCESS;
  }

  // ------------------------------------------------------------------
  // Print execution options
  // ------------------------------------------------------------------
  void print_info(void)
  {
    std::cout << "KMC dump ver. " << KMC_VER << " (" << KMC_DATE << ")\n";
    std::cout << "\nUsage:\nkmc_dump [options] <kmc_database> <output_file>\n";
    std::cout << "Parameters:\n";
    std::cout << "<kmc_database> - kmer_counter's output\n";
    std::cout << "Options:\n";
    std::cout << "-ci<value> - print k-mers occurring less than <value> times\n";
    std::cout << "-cx<value> - print k-mers occurring more of than <value> times\n";
  };
```

Figure 2: Second part of kmc_dump_sample application

3 Database format

The KMC application creates output files with two extensions:

- .kmc_pre — with information on k-mer prefixes (plus some other data),
- .kmc_suf — with information on k-mer suffixes and the related counters.

All integers in the KMC output files are stored in LSB (least significant byte first) byte order.

3.1 The .kmc_pre file structure

The .kmc_pre file contains, in order, the following data:

- [marker],
- [prefixes],
- [map],
- [header],
- [header position],
- [marker] (another copy, to signal the file is not truncated).

[marker]

4 bytes with the letters: KMCP.

[header position]

The integer consisting of the last 4 bytes in the file (before end KMCP marker). It contains the relative position of the beginning of the field [header]. After opening the file, one should do the following:

1. Read the first 4 bytes and check if they contain the letters KMCP.
2. Read the last 4 bytes and check if they contain the letters KMCP.
3. Jump to position 8 bytes back from end of file and read the header position x.
4. Jump to position $x + 8$ bytes back from end of file and read the header.
5. Read [data].

[header]

The header contains fields describing the file .kmc_pre:

- uint32 kmer_length — k-mer length,
- uint32 mode — mode: 0 (occurrence counters) or 1 (quality-aware counters),
- uint32 counter_size — counter field size: for mode 0 it is 1, 2, 3, or 4; for mode 1 it is always 4,

- uint32 lut_prefix_length — the length (in symbols) of the prefix cut off from k-mers; it is invariant of the scheme that 4 divides $(kmer\_length - lut\_prefix\_length)$,
- uint32 signature_length — the length (in symbols) of the signature,
- uint32 min_count — minimum number of k-mer occurrences to write in the database (if the counter is smaller, the k-mer data are not written),
- uint32 max_count — maximum number of k-mer occurrences to write in the database (if the counter is greater, the k-mer data are not written),
- uint64 total_kmers — total number of k-mers in the database,
- uchar both_strands — 1 if KMC was launched with -b switch, 0 otherwise,
- uchar tmp[3] — not used in the current version,
- uint32 tmp[6] — not used in the current version,
- uint32 KMC_VER — version of KMC software (for KMC 2 this value is equal to 0x200).

[map]

There is an array of uint32 elements, of size $4^{signature\_length} + 1$. This array is used to identify position of proper prefixes' array stored in [prefixes] region. For example, if the queried k-mer is ATACGACAAATG and $signature\_length = 5$, its signature is ACGAC (as it is the smallest 5-mer which satisfies conditions of being a signature). DNA symbols are encoded as follows: A $\rightarrow$ 0, C $\rightarrow$ 1, G $\rightarrow$ 2, T $\rightarrow$ 3, so ACGAC is equal to 97 (since $0 \cdot 2^8 + 1 \cdot 2^6 + 2 \cdot 2^4 + 0 \cdot 2^2 + 1 \cdot 2^0 = 97$). In this case we look into "map" at position 97 to get the id of related prefixes' array.

[prefixes]

This region contains a number of prefixes' arrays (typically hundreds of them) of uint64 elements. Each array is of size $4^{lut\_prefix\_length}$. The last prefixes' array is followed by an additional uint64 element being a guard to make the reading process simpler. The total number of prefixes' arrays can be easily calculated (as start and end position are given, size of one array is also known). The element at position x in prefixes' array for given signature s points to a record in .kmc_suf file. This record contains the first suffix of k-mer with prefix x and signature s (the position of the last record can be obtained by decreasing the value at $x + 1$ in prefixes' array by 1).

Using the example from the previous section, the start position of prefixes' array for k-mer ATACGACAAATG should be calculated as: $4 + 97 \cdot 4^{lut\_prefix\_length} \cdot 8$ (marker + equivalent of ACGAC signature · no. of elements in each array · size of element in prefix array). The next step is to cut off the prefix of length equal to lut_prefix_length from the queried k-mer. Let us assume $lut\_prefix\_length = 4$, and then the prefix is ATAC whose equivalent is 49. The element at position 49 in the related prefixes' array (pointed by signature 97) is the position of the first record in .kmc_suf file which contains a k-mer with prefix ATAC and with signature ACGAC. Let us suppose this position is 1523, then we look at position 50 in prefixes' array (say, it contains 1685). This means that .kmc_suf file stores the suffixes of k-mers with prefix ATAC and signature ACGAC in the records from 1523 to $1685 - 1$. Having got this range, we can now apply binary search for the suffix GACAAATG.

3.2 The.kmc_suf file structure

The .kmc_suf file contains, in order, the following data:

- [marker],
- [data],
- [marker] (another copy, to signal the file is not truncated).

The k-mers are stored with their leftmost symbol first, packed into bytes. For example, CCACAAAT is represented as 0x51 (for CCAC), 0x03 (for AAAT). Integers are stored according to the LSB (little endian) byte order, floats are stored in the same way as they are stored in the memory.

[marker]

4 bytes with the letters: KMCS.

[data]

An array record_t records[total_kmers].

total_kmers is taken from the .kmc_pre file.

record_t is a type representing a k-mer. Its first field is the k-mer suffix string, stored on $(kmer\_length - lut\_prefix\_length)/4$ bytes. The next field is $counter\_size$, with the number of bytes used by the counter, which is either a $1 \ldots 4$-byte integer, or a 4-byte float.