# The Sixth DIMACS Implementation Challenge:

## Near Neighbor Searches

http://www.dimacs.rutgers.edu/Challenges/Sixth/

## Optional Development Guide
(version 1.0)

Michael Goldwasser
challenge6@dimacs.rutgers.edu

January 16, 1998

## 1 Disclaimer

Participants in the Implementation Challenge are free to develop their own software, using whatever language, format, and structure they wish. However in order to allow better co-operation between participants in the Challenge, we have developed the following suggested "standards" for software development. The purpose of these standards is to allow participants to freely interchange algorithms and data sets with each other, thereby allowing people to test various algorithms on various data sets interchangeably.

## 2 Overview of Interface

There are two distinct tracks for participation in this year's Implementation Challenge on near neighbor searches. The first is to design and implement algorithms for various nearest neighbor searchers, however the second is to gather interesting examples of data sets and distance functions to use as input for the algorithms. Most years in the Challenge, a standard file format for the input is agreed upon so that all algorithms can be run on a testbed of inputs. If we were only considering Euclidean metric spaces, for example, it would be fairly simple to design a file format which lists the vectors for the original and query points. However, we want to also consider algorithms for general spaces, which may measure similarity of web documents, images, DNA sequences, lines of text, and so on. The only common input form for all of these data sets may be to simply provide the entire distance matrix for all points and queries. Unfortunately, our aim is to have efficient algorithms even for very large data sets which are encountered in today's applications, and so providing the entire distance matrix seems impractical.

For this reason, we have elected to take a different approach this year. We have designed an interface in which distance oracles for data sets will be designed as completely separate programs from the neighbor search algorithms, and we will have the two programs communicate through a client/server model using Internet/Unix sockets. In this way each search program and distance oracle can be designed and compiled separately, and so the same search program can be tested on a variety of input sets based on very different underlying domains. A program which acts as an oracle for a data set must simply be prepared to answer a variety of queries regarding the underlying space, and a program for a search algorithm will get all of its information about the data by calling similar query functions. We will be providing sample drivers written in C, which make this communication as transparent as possible for each side. The initial implementation uses UDP, but we hope to release a more robust implemenation soon which allows a choice of TCP/UDP interfaces as well as the use of UNIX sockets. For those who wish, the communication routines can be removed, and the algorithm and oracle can be combined together and compiled as a single process (see Section 6).

We see many potential advantages to this clear separation in the design specifications. For starters, this solves the initial problem of providing a clean interface for those wishing to implement one side of the Challenge without concern for the other. Since both ends will run as separate processes, there is no need to recompile any program in order to mix and match algorithms and input sets. Also, the specification for the oracle will allow for a single data set oracle to be left running as a server, answering queries for many different search algorithms at the same time. This may be especially useful for domains in which the loading time is quite large for an oracle. Finally, although commonly the algorithm program and the oracle program will be running on the same machine, because the client/server interface uses messages over the Internet, there really is no requirement that the two ends be running on the same machine, using the same programming language, or even on the same platform. Programs could conceivably be designed in completely different languages, or on different platforms, so long as both ends follow the agreed upon protocol for passing inquiry messages and responses. Furthermore, we envision the possibility of someone providing an oracle running over the Internet, without necessarily publicly releasing the underlying data or the source code used for distance comparisons. This could be done by simply announcing the Internet address and port at which the oracle is left running.

### 2.1 The rest of this file

The rest of this document will serve as a detailed programmer's guide for all aspects of the interface for both search algorithms as well as data set oracles. Section 3 will discuss the interface for writing neighbor search algorithms, whereas Section 4 will discuss the interface for writing data set oracles. Section 5 discusses how application-specific information can be passed from an oracle to a specialized algorithm for that domain. For instance, for Euclidean metric spaces, algorithms may need to know the coordinates for the underlying points, not simply the distances between points. For those who wish to remove the use of Internet messages for communication, Section 6 discusses the details of how an algorithm and data set can be combined and compiled as one program. The majority of this document will be based on the assumption that programs are designed for the C programming language, however Section 7 will discuss the possibility of using other languages in conjunction with these specifications. All of the software examples discussed in this document can be downloaded directly from the software section of the Challenge web page.

## 3 Neighbor Search Algorithms

Our specification for an implementation of a search algorithm consists of the following headers:

- `algorithm.h` — This header file contains the function specifications for the heart of the near neighbor search algorithm. We have implemented a naive brute force search algorithm in algorithm.c, however this is the main file you will want to rewrite.

- oracle.h — This header file contains the communication functions used to make inquiries from the data set oracle. You will want to call these functions as part of your near neighbor search algorithm. In the client/server model, these functions will be implemented by the low-level network routines in client.c which will send a message to the oracle.
- client.h — This header file simply contains to routines for opening and closing the connection to the oracle.
- search.c — This contains the main routine. We have written a simple front end driver, which establishes a connection with the oracle and controls the high-level flow of control for an experiment. The experiment we have simply allows the algorithm to preprocess the data set, and then gives it a series of neighbor queries. This file could be changed to perform a different experiment flow if desired.

### 3.1 Routines which you must implement

To provide a search algorithm as part of the Implementation Challenge, you must implement the following routines, whose prototypes are given in the file algorithm.h. Each of these routines refers to the data type AbstractDS. From the outside world view, this data type is simply a void, used as a handle to pass between routines. Within the implementation this will be defined as whatever data structure you choose for your algorithm. The functions which you must implement are:

```
/******************************************************
 * AbstractDS* Preprocess(Oracle* ora)
 *
 * Parameters:
 *   Oracle* ora --- pointer to the data set oracle
 *
 * Returns:
 *   A pointer to a data structure of your choice which is passed on
 *   to the routine 'FindNeighbor' whenever a later query is made.
 *
 * Description:
 *   The purpose of this routine depends greatly on the nearest
 *   neighbor algorithm which will be used to answer queries.  Access
 *   to the data set is given through the use of the oracle.  All
 *   preprocessing should take place in this routine, and any
 *   resulting data structures must be saved as part of the returned
 *   object.
 *
 * Note:
 *   This routine should NOT ask the oracle about any of the query points.
 * Additional Note:
 *   If the algorithm requires additional parameters from the user,
 *   These should be added to the functions parameter list, and the
 *   corresponding values should be sent by the main program when
 *   calling this function.
 ******************************************************/
```

```
/*****************************************************
 * void FindNeighbor(Oracle* ora,  AbstractDS* ads, int query)
 *
 * Parameters:
 *   Oracle* ora     --- pointer to the data set oracle
 *   AbstractDS* ads --- pointer to your data structure
 *   int query       --- which query point
 *
 * Description:
 *   This is the neighbor search query routine.  The algorithm may use
 *   the preprocessed data structure in conjunction with questions to
 *   the oracle about the query point.  Any output or responses to the
 *   query should be made from within this routine.
 *
 * Note:
 *   If theory, you could consider query algorithms which alter the
 *   preprocessed data structure to reflect any new and useful
 *   knowledge.
 *
 *****************************************************/

/*****************************************************
 * void DestroyAbstractDS(AbstractDS* ads)
 *
 * Parameters:
 *   AbstractDS* ads --- pointer to your data structure
 *
 * Description:
 *   This is simply a cleanup routine.
 *
 *****************************************************/

/*****************************************************
 * int CostPreprocess(AbstractDS* ads)
 *
 * Parameters:
 *   AbstractDS* ads --- pointer to your data structure
 *
 * Returns:
 *   An integer value which represents the 'cost' of the
 *   preprocessing step, in whatever units you choose to report.
 *
 * Note:
 *   The accounting should have been done during the preprocessing
 *   step, with the cost value saved as part of the data structure, in
 *   order to report it at the time of this procedure call.
 *
 *****************************************************/
```

```
/*****************************************************
 * int CostQueries(AbstractDS* ads)
 *
 * Parameters:
 *   AbstractDS* ads --- pointer to your data structure
 *
 * Returns:
 *   An integer value which represents the 'cost' of the combined
 *   total of all neighbor queries.
 *
 * Note:
 *   The accounting should have been done during each of the neighbor
 *   queries, with the cost value saved as part of the data structure,
 *   in order to report it at the time of this procedure call.
 *
 *****************************************************/
```

### 3.2 Communicating with the oracle

During your implementation of `Preprocess` and `FindNeighbor`, you will of course need to communicate with the oracle to be able to get information about the data set. For those programming in C, we have provided routines, described in `client.h`, which handle the network communication with the oracle transparently. Before using any of the inquiry routines, it is necessary to first call the function `OracleConnect` which establishes a connection with the oracle server, and returns a symbolic pointer to the oracle which is required for the remaining routines. If you use our `search.c` driver, this connection will be established for you.

The main functionality provided by the oracle is the `InqDist` function which provides a real-valued distance between any two points of the data set. This function is used, both to compare two points from the original space, as well as to compare a query point to points in the original space. Our convention is that all of the points of the space, including the query points, are given unique integer indices, starting at zero, where the query points are indexed after all of the original points. For example, if the space consists of 1000 original points, and 100 other query points, then the original points will be numbered from 0 to 999, followed by the query points from 1000 to 1099.

Although points of various data sets represent objects from varied domains, we assume that each point has some associated data which may be passed to a specialized algorithm. Each data point may have an arbitrary number of associated fields, where each field is a string. In general, the content of this string is purely up to the discretion of the oracle, as it will only be echoed as a more descriptive name for data points, or will be used by search algorithms which know the structure of the data and have been specialized to work with a particular type of data sets. Specific protocols for these fields will be discussed in Section 5 .

The full set of functions available are specified in `oracle.h` as follows:

```
/*****************************************************
 * int InqNumPoints(Oracle *ora)
 *
 * Parameters:
 *   Oracle* ora --- an oracle connection
 *
```

```
 * Returns:
 *   The number of points in the original data set
 *
 * Description:
 *   If the routine returns 'n' this means that there are n points in
 *   the data set (not counting any queries).  When querying the
 *   oracle about individual points, they are referenced with IDs
 *   ranging from 0 to n-1.
 *
 *******************************************************/

/********************************************************
 * int InqNumQuery(Oracle *ora)
 *
 * Parameters:
 *   Oracle* ora --- an oracle connection
 *
 * Returns:
 *   The number of query points available
 *
 * Description:
 *   If the routine returns 'q' this means that there are q query
 *   points available.  When querying the oracle about these points,
 *   we assume that they are given IDs consecutively following all of
 *   the original data points.  That is, if there are 'n' original
 *   points and 'q' query points, the query points are indexed in the
 *   range [n,n+q-1].
 *
 *******************************************************/

/********************************************************
 * double InqDist(Oracle *ora, int p1, int p2)
 *
 * Parameters:
 *   Oracle* ora --- an oracle connection
 *   int p1      --- index to first point
 *   int p2      --- index to second point
 *
 * Returns:
 *   The non-negative, real-numbered, distance between the points
 *
 * Description:
 *   Each point can be either an original point in the data set,
 *   or one of the query points.  The convention for indexing is
 *   identical to the above functions.
 *
 *******************************************************/
/********************************************************
```

```
 * int InqNumFields(Oracle *ora, int p)
 *
 * Parameters:
 *   Oracle* ora --- an oracle connection
 *   int p       --- index to data point
 *
 * Returns:
 *   An integer specifying how many fields describe the
 *   underlying raw data for point p. (This could potentially be
 *   a different value for different points in the underlying space.
 *
 * Description:
 *   The point can be the index for either an original point in the
 *   underlying space, or one of the query points.  The convention for
 *   indexing points is identical to the above functions.  The
 *   convention for indexing fields is from [0..InqNumField-1].
 *
 * Note:
 *   In the case of Euclidean vectors, this string should be the
 *   number of dimensions.
 *
 *******************************************************/

/*******************************************************
 * char* InqField(Oracle *ora, int p, int f)
 *
 * Parameters:
 *   Oracle* ora --- an oracle connection
 *   int p       --- index to data point
 *   int f       --- which field of raw data
 *
 * Returns:
 *   A string of length at most MAX_FIELD_LENGTH which is the
 *   associated field for the raw data describing a point in the
 *   space.  The ownership of this string is passed to the caller, and
 *   it is assumed that the caller will free the memory for it.
 *
 * Description:
 *   The point can be the index for either an original point in the
 *   underlying space, or one of the query points.  The convention for
 *   indexing points is identical to the above functions.  The
 *   convention for indexing fields is from [0..InqNumField-1].
 *
 * Note:
 *   In the case of Euclidean vectors, this string should be the
 *   real-valued entry for the coordinate in question.
 *******************************************************/
```

# 4 Data Set Oracles

The details of implementing a data set oracle depend heavily on the exact domain of the underlying space, as well as on the method for generating or reading input. We have divided the implementation into the following three pieces:

- `main driver` – You will have to implement the main driver which first loads or generates the data points and queries for the problem instance. We have implemented two sample oracles: *randcube.c*, which generates random vectors from a cube in Euclidean space; *text.c*, which uses lines of texts as data and generates a metric space based on one of several edit distance measures.
- `oracle.h` — This header file contains the functions which will be asked of your oracle by search algorithms. You must implement these functions, although this may be done in the same file as the main driver.
- `server.h` — This header file contains the network functions used to set up a server. You will need to call these functions after loading the data set. We have implemented the low-level network routines in `server.c`, and there should be no need for you to alter this file.

## 4.1 Top Level Control

We ask that you follow the general flow of control described here. First, you should generate a *static* data structure which represents your entire original data set, as well as all of the query points which will be used. When creating your data, you should conventionally number all points, including your query points, so that each is given a unique integer index, starting with zero, and where the query points are indexed after all of the original points. For example, if the space consists of 1000 original points, and 100 other query points, then the original points will be numbered from 0 to 999, followed by the query points from 1000 to 1099. Furthermore, you may assign each point associated data fields of your choice. In general, these string have no use for a near neighbor algorithm other than perhaps descriptive output, however specialized algorithms may be developed which rely on knowing the underlying data. The maximum length of any given field is currently set to 50 in `oracle.h`. In the special case of Euclidean metric spaces, these associated string will be the actual coordinate values (see Section 5).

Once this setup is complete, you should call the routine `ServerOpen` to establish yourself as a server and begin waiting for query messages from clients. When opening a server, you must pass it a pointer to your own data structure. This pointer will later be passed on as a parameter to any event handlers for incoming message queries, and so this data structure must have all information needed by your message handlers. Notice that because you have generated a static data structure in advance, your server will actually be able to handle interleaved queries from many clients at once. The exact specification for `ServerOpen` is given as:

```
/*******************************************************
 * void  ServerOpen(int port, Oracle* ora, int quiet)
 *
 * Parameters:
 *   int port         --- what port should server use
 *   Oracle* ora      --- pointer sent to even handlers
 *   int quiet        --- should server echo messages/responses to screen
 *
```

```
* Returns:
*   A symbolic pointer to the new server.
*
* Description:
*   This routine starts up a server on the local machine, at the
*   specified port.  The Oracle pointer is simply held, and then
*   sent on to any event handlers when query messages arrive.
*
*******************************************************/
```

### 4.2 Event Handlers

In designing an oracle, you must implement event handlers for answering inquiries from algorithms. Every time an incoming message arrives at the server for a particular query, the appropriate event handler will be called automatically. The first parameter sent to each of these handlers is exactly the data structure pointer which you sent to the `ServerOpen` call. In the event that a message arrives asking about points which are not within the correct range, your response is irrelevant, however you should make sure that such bogus queries do not crash your oracle. The interface for the event handlers, given in `oracle.h`, is the same as the interface used by search algorithms and is provided in Section 3.2.

## 5 Associated Data Fields

Some algorithms will be able to operate without knowing anything about the underlying data sets other than simply the distance between various points. On the other hand, more specific algorithms may exists which are fine-tuned for specific application domains. For example, a search algorithm for Euclidean space may use geometric techniques to improve the performance, yet must have access to the underlying vectors for each point. It is for exactly this reason that we have added the `InqNumFields` and `InqField` routines to the oracle interface. These routines may always be ignored by a search algorithm, however this will allow us to pass needed information between the oracles and algorithms for specialized instance domains.

We suggest the following protocols for the use of the associated data fields, and we hope to have more added by participants as time goes (see the web page for newer protocols).

### 5.1 Euclidean Metric Spaces

The associated data for a point in a $d$-dimensional Euclidean metric space should consists of exactly $d$ fields, one for each dimension, numbered from $[0..d-1]$. The string returned by `InqField(ora,p,i)` should be a floating point representation for the $i$th coordinate of point $p$ (currently with maximum precision of 50 characters). A sample of this protocol is given in the oracle `randcube`, available from the Challenge web page.

### 5.2 Color-coded Classification

One of the possible areas of research involves the case where points of the data set are color-coded, and queries are asked not about the identity of the neighbors, but simply the colors of the neighbors (see Participation Overview document for more details).

Our protocol for color-coded sets will be to have the first field equal to the color, specified as an identifying integer. Notice, that this protocol can be placed on top of other protocols for color-coded versions of the protocols by using the first field to be the color, and offsetting all other fields by one. For example, a color-coded data set for $d$-dimensional Euclidean space would now have $(d+1)$ fields, the first of which is equal to the color, and the remaining fields equal to the vector coordinates.

## 6 Avoiding the Client/Server Communication

In some cases, you may wish to develop your programs without the use of message passing between oracles and algorithms on the Internet (for example, if running on a machine not on the Internet). With only a slight modification, this interface can be modified to allow you to combine the algorithm and oracle into a single program. To do so, combine the files from an algorithm implementation and an oracle implementation into the same directory (two files will be in common). Then, simply delete the files `server.c`, `server.h`, `client.c`, and `client.h`. The algorithms call to the oracle, will now be answered directly by the oracle itself. The only necessary modification to the code will be to combine the original two main routines into one unified routine, as there will now be a single process. Unfortunately, to run an algorithm with a different oracle implementation requires an entirely new compilation.

## 7 Using Other Languages and Platforms

There is nothing stopping you from using whatever language or platform you wish, so long as you know more about clients, servers, and sockets than I do. For those programming in C (and hence C++), we have put together all of the low-level software which handles the communication between the two processes. To use some other language it will be necessary to either link in the C routines for communication, or else to redesign similar communication routines in the other language.

For sake of completeness, we summarize the underlying message protocol used by both ends, and defined in the `message.h` file. After establishing a connection, the client side process initiates all queries, sending a message to the oracle query, and waiting for a response from the oracle.

```
/*******************************************************
 *  Query:  How many data points are in the original data set?
 *  Client Message:  "P"
 *  Server Response: "<num>",  where num is an integer
 *******************************************************/

/*******************************************************
 *  Query:  How many query points are available?
 *  Client Message:  "Q"
 *  Server Response: "<num>", where num is an integer
 *******************************************************/

/*******************************************************
 *  Query:  What is the distance between points i and j?
 *  Client Message:  "D <i> <j>"
 *  Server Response: "<dist>", where dist is a floating point
```

```
 *****************************************************/

/*****************************************************
 *  Query:  How many fields in the associated data for point i?
 *  Client Message:  "A <i>"
 *  Server Response: "<string>"
 *****************************************************/

/*****************************************************
 *  Query:  What is the associated data in field f, for point i?
 *  Client Message:  "F <i> <f>"
 *  Server Response: "<string>"
 *****************************************************/

/*****************************************************
 *  Unknown Query:  If the server ever receives a query which does not
 *  match one of the above, rather than not respond, it responds that
 *  it received an unknown query as follows.
 *
 *  Server Response: "U"
 *****************************************************/
```