WANN: AN IMPLEMENTATION OF WEIGHTED NEAREST NEIGHBOR SEARCH

A. ANDREIVSKIĬ AND A. SOBOLEVSKIĬ

Abstract. WANN (Weighted Approximate Nearest Neighbors) is a library of C++ classes for weighted nearest neighbor search. It is based on the ANN library of D. Mount and S. Arya and generalizes their implementation of nearest neighbor search, providing a building block for solution of the discrete transport problem. WANN is meant as part of a multipurpose library for numerical transport optimization. Installation and basic usage of the WANN library are described.

1. Introduction

The standard *nearest neighbor problem* is formulated as follows: given a finite set of points P in d-dimensional Euclidean space and a query point q, find the point $p^*(q)$ in P whose distance from q is minimal:

$$p^*(q) = \arg\min_{p\in P} |q-p|^2. \tag{1}$$

Minimizing the square of distance, rather than distance itself, is immaterial for the nearest-neighbor search as such, but squared distances are significant for the application of a generalized version of this problem to reconstruction of cosmological velocity and density fields from galaxy catalogues [9, 7, 8] by the Monge–Ampère–Kantorovich (MAK) method [6, 2] and, in a wider context, for various numerical transport optimization techniques.

Let now Q be a set of query points with $1 \le |Q| \le |P|$. The *discrete transport problem* is to find an invertible map π from Q to P that minimizes the quantity

$$\sum_{q\in Q} |q-\pi(q)|^2. \tag{2}$$

The problem is called *asymmetric* if $|P| \ne |Q|$; when $|P| = |Q|$ it becomes an instance of the *assignment problem.* Minimizing (2) with no constraints on π can be reduced to a collection of $|Q|$ independent nearest neighbor problems; the nearest neighbor problem (1) is recovered when $|Q| = 1$.

1.1. **Why weighted?**

For small $|P|$ and $|Q|$ solution may be obtained by computing sums of squared distances (2) for each of the $|P|!/(|P|-|Q|)!$ possible choices of the map π. A more sophisticated approach is based on the following idea. Suppose at each q in Q there is a little creature that wants to move to a certain p in P but has in the process to pay a *transport cost* proportional to $|q-p|^2$. Each creature would like to go to its nearest neighbor in P, but if a few creatures compete for the same p, only one can succeed. To settle the conflict, the creatures pay not only the transport cost but an additional fee $w(p)$ for being admitted to a specific p, which may make distant points more attractive than nearest neighbors. It is not difficult to show (see e.g. [1]) that there exists a set of *equilibrium* fees w^* such that the map π^* minimizing (2) reduces to solving $|Q|$ independent minimization problems of the form

$$\pi^*(q) = \arg\min_{p\in P}\bigl(|q-p|^2 + w^*(p)\bigr). \tag{3}$$

In this note the fees are called *weights* and the problem of minimizing the expression in the r.h.s. of (3), the *weighted nearest neighbor problem*.

1.2. **Why a dedicated implementation?**

The discrete transport problem is an instance of network flow problem, itself a particular case of the general linear programming problem. For these classes of optimization problems there is a large supply of efficient algorithms and fast solvers. In our terms these algorithms often involve a sequence of weighted nearest neighbor searches alternated with suitably updating the weights.

For the discrete transport problem, a generic linear programming solver would require $O(|P|\,|Q|)$ space for the cost matrix $c(p,q) = |p-q|^2$ as well as for the solution, which is expressed as a *transport plan* $x(p,q)$ prescribing how much mass goes from p to q. However an optimal transport plan uses only $O(\max(|P|,|Q|))$ variables (those for which $p = \pi^*(q)$). Dedicated network flow solvers take into account the sparsity of the transport plan, but often fail to recognize the effective sparsity of $c(p,q)$: the elements of this matrix can be computed on demand from the $O(|P|+|Q|)$ storage necessary for the sets P and Q rather than precomputed and stored. More importantly, where the weighted nearest-neighbor search is required in the discrete transport problem, these algorithms typically perform inefficient brute-force search if the cost matrix is full.

For these reasons, it is of interest to develop a dedicated solver for discrete transport problems based on adequate geometric search routines. This note describes an implementation of these routines.

1.3. **Why ANN?**

There are quite a few nearest neighbor solvers based on preprocessing the set P into various data structures that allow to efficiently find nearest neighbors; see e.g. [11, 3, 5], the STANN project [4], and the links at the Stony Brook Algorithm Repository at [13, Section 1.6.5]. One of the most common approaches to the problem is based on preprocessing the set P into a kd-tree. There are many flavors of kd-trees, described e.g. in the ANN Programming Manual [10] (see also a chapter on kd-trees in Numerical Recipes [12]).

In a nutshell, a kd-tree is a binary tree whose root is the bounding box for the set P and whose nodes correspond to nested boxes with edges paralles to coordinate axes; each box contains a portion of the point set P and each of its two subboxes contains approximately one half of that portion. Nearest neighbor search in a kd-tree is based on the fact that the distance from a query point to any point contained in a box can be estimated from below with the distance from the query point to this box. After a candidate point is found, this simple property allows to discard boxes that are farther from the query point than the candidate, and to achieve a typical performance of $O(\log|P|)$ operations per query, where $\log|P|$ corresponds to the depth of the kd-tree. The worst case behaviour of this data structure may be significantly worse but in typical discrete transport applications in low-dimensional spaces one should not worry about it.

Of all the approaches to nearest neighbor search listed above, kd-trees are probably best suited for the weighted nearest neighbor search. Indeed, if one maintains for each box the minimal weight of its points, then the total cost for points in this box can again be estimated from below with the sum of the squared distance to the box and this minimal value, allowing to discard boxes when processign a query. Resetting a point's weight may require to update minimal weight values of its parent boxes, but this is achieved, too, in $O(\log|P|)$ operations.

The most well-known and stable implementation of nearest neighbor search based on kd-trees is arguably the ANN library by D. Mount and S. Arya [11, 10]. In the present note we report an extension of this library to weighted nearest neighbor search. Care was taken to fully retain the original ANN functionality: the WANN library may even be compiled without the support for weights, which makes it identical to ANN (see subsection 2.1 on the build process below).

2. Using WANN

2.1. **Downloading and buiding WANN.**

The current version of WANN is 0.2; is has the status of a beta release. The WANN package including the source code and documentation is availbale at the web page

`http://www.mccme.ru/~ansobol/otarie/software.html`

Most of the directory layout in WANN version 0.2 comes from the the original ANN distribution. After unpacking the package a directory `wann-0.2` will be created containing the following directories:

- `include/`: Include files for compiling programs that use the WANN library.
- `src/`: The source files for the WANN library.
- `sample/`: A small sample program that shows how to use the WANN library.
- `test/`: The program `ann_test` that provides a simple script language for preprocessing point sets and performing weighted nearest-neighbor queries.
- `ann2fig/`: The program `ann2fig` that generates a visual representation in fig format for the preprocessed set.
- `bin/`, `lib/`: These directories are used by the original ANN make script to put binaries and the library file built from ANN sources. They are not needed for the WANN build procedure and may be removed from later versions of the WANN package.
- `doc/`: Contains the ANN Programming manual [10], the WANN documentation (this file), and the sample program `wanntest.cxx` described in subsection 2.2.
- `MS_Win32/`: Solution and project for compiling the original ANN library under Microsoft Windows (TM).

For details on how to use the `ann_test` and `ann2fig` programs see the ANN Programming Manual [10].

To build and install the WANN library on your system, descend to the directory `wann-0.2/` and issue the following commands: `./configure`; `make`; `[sudo] make install`. Running `make install` requires root privileges, and `sudo` is a way of obtaining them on BSD-like Unices. Check documentation to your system to find out how to obtain root privileges (or run `make install` from a superuser account).

The basic build procedure described in the previous paragraph will build WANN as both static and (system permitting) shared library, and will also build and install the `ann_test` and `ann2fig` utilities. Performance evaluation will not be enabled. To customize the build procedure use the following options to the `configure` script:

- `--disable-weights`: Build the WANN library *without* weighted nearest neighbor search (so that it becomes identical to the original ANN library of D. Mount and S. Arya).
- `--disable-ann_test`: Do not build the `ann_test` utility.
- `--disable-ann2fig`: Do not build the `ann2fig` program.
- `--enable-stats`: Enable performance evaluation (see the ANN Programming Manual [10] for a detailed description).

The `make install` command will install the WANN library and the `ann_test` and `ann2fig` utilities into subdirectories of the default system locations (on most Unices they are `/usr/local/include/` for libraries and `/usr/local/bin/` for executables). These locations can be changed in the invocation of the `configure` script. E.g., to install WANN to directories under your home directory run the script with `./configure prefix=$HOME`.

Help on customization of the configuration and build procedure can be obtained by running `./configure --help`.

Care was taken in this release to ensure as much compatibility with the original ANN library as possible. In particular, there is a possibility to run the original ANN build procedure, which is not based on the `configure` script. In order to do this, run the `unWANN` script in the `wann-0.2/` directory, which moves the files named `Makefile.ANN` to `Makefile` in all subdirectories, renames `include/wann/` to `include/ANN/`, edits the file `include/ANN/ANN.h` to define the `ANN_NO_WEIGHTS` symbol, and edits WANN source files to reflect renaming the include directory. After this the `make` command will invoke the original ANN build procedure and the compiled library may be used exactly as if it were ANN. Note that using this feature is deprecated and may be not supported in later versions of the WANN library.

2.2. **A sample program.**

Here is a sample program `wanntest.cxx` illustrating how to use the weighted nearest neighbor search.

```
#include <wann/ANN.h>
#include <iostream>

#define N   3
#define DIM 2
#define K   1

using namespace std;

int main ()
{
    ANNpointArray dataPts;
    ANNdistArray  dataWeights;
    ANNpoint      q;
    ANNidxArray   nnIdx;
    ANNdistArray  costs;
    ANNkd_tree*   kdTree;
18
19      q           = annAllocPt(DIM);
20      dataPts     = annAllocPts(N, DIM);
21      dataWeights = annAllocWeights(N);
22      nnIdx       = new ANNidx[K];
23      costs       = new ANNdist[K];
24
25      dataPts[0][0] = -1.; dataPts[0][1] =  0.;
26      dataPts[1][0] =  1.; dataPts[1][1] = -1.;
27      dataPts[2][0] =  1.; dataPts[2][1] =  1.;
28      dataWeights[0] = dataWeights[1] = dataWeights[2] = 0.;
29
30      kdTree = new ANNkd_tree(dataPts, dataWeights, N, DIM);
31
32      q[0] = q[1] = 0.;
33
34      kdTree->annkSearch(q, K, nnIdx, costs, 0.);
35
36      cout << "Nearest neighbor: (";
37      cout << dataPts[nnIdx[0]][0] << ", ";
38      cout << dataPts[nnIdx[0]][1] << "); squared dist = ";
39      cout << costs[0] << "\n";
40
41      kdTree->setWeight(0, 2.);
42
43      kdTree->annkSearch(q, K, nnIdx, costs, 0.);
44
45      cout << "Weighted nearest neighbor: (";
46      cout << dataPts[nnIdx[0]][0] << ", ";
47      cout << dataPts[nnIdx[0]][1] << "); squared dist = ";
48      cout << costs[0] - dataWeights[nnIdx[0]] << "\n";
49
50      annDeallocPt(q);
51      annDeallocPts(dataPts);
52      annDeallocWeights(dataWeights);
53      delete [] nnIdx;
54      delete [] costs;
55      delete kdTree;
56      annClose();
57
58      return 0;
59  }
```

The WANN library header file is included in line 1 (its name is unchanged from the original ANN implementation). In lines 4–6, there are defined the number of points `N` $= 3$, the dimension of space `DIM` $= 2$, and the number of nearest neighbors to look for `K` $= 1$. The arrays `nnIdx` and `costs`, which in our case both have only one element, contain indices of optimal points and values of the transportation cost from the query point.

The array `dataPts` contains coordinates of three points in two-dimensional plane: $(-1, 0)$, $(1, -1)$, and $(1, 1)$. The array is filled in lines 25–27; normally the data points would be read from an input file or generated by a separate routine. Weghts of all three points are contained in the array `dataWeights` initially filled with zeros in line 28. The array `q` contains coordinates of the query point, $(0, 0)$.

The main data structure, a kd-tree `kdTree`, is built in line 30. The search routine is called twice. In line 34 it returns the nearest neighbor of the query point, $(-1, 0)$. After this the weight of this point is set to 2, making the other two points optimal in the weighted nearest neighbor problem. Of these, the last one is picked up by the search routine.

Note the explicit deallocation of all dynamically allocated data in lines 50–56.

This program has been compiled on a Mac OS X 10.5 system with the command

```
g++ -L/usr/local/lib/wann -lWANN wanntest.cxx
```

The `-L` option specifies the path to the directory where the compiled WANN library file `libWANN.a` is located; as is standard for a GCC compiler, the `-l` (small ℓ) option gives the name of the library file without the prefix `lib` and the suffix `.a`. Here is the output:

```
Nearest neighbor: (-1, 0); squared dist = 1
Weighted nearest neighbor: (1, 1); squared dist = 2
```

If the WANN library has been compiled with weights disabled but using the normal WNN build brocedure (as opposed to the deprecated ANN build procedure set up by the `unWANN` script), then the following line must be added in front of including `ANN.h`:

```
#define ANN_NO_WEIGHTS 1
```

If the WANN library has been compiled with toric geometry enabled, then the following line must be added in front of including `ANN.h`:

```
#define WANN_TORUS 1
```

Note that in the present version, there is no way of inputting torus sizes to the program `ann_test`: torus is hard-coded to have unit sizes of all dimensions. It is the user's responsibility to supply point coordinates inside the unit torus. Of course this will change in a future version.

Acknowledgments

We are grateful to U. Frisch and R. Mohayaee for initiating the cosmological application of optimal transport, to M. Hénon for sharing his early work on the discrete transport problem, and to J. Matoušek for a pointer to kd-trees as efficient means of search in Euclidean space, which eventually led to the approach presented here. Special thanks go to David M. Mount and Sunil Arya for the development of the ANN library. This work is supported by the ANR (Agence nationale de la recherche, France) under grant BLAN07-2_183172 OTARIE and by the RFBR (Russian Foundation for Basic Research) under grant RFBR/CNRS 07–01–92217.

References

[1] Bertsekas, D. P. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications 1*, 1 (1992), 7–66.

[2] Brenier, Y., Frisch, U., Hénon, M., Loeper, G., Matarrese, S., Mohayaee, R., and Sobolevskiĭ, A. Reconstruction of the early Universe as a convex optimization problem. *Monthly Notices of the Royal Astronomical Society 346*, 2 (December 2003), 501–524.

[3] Chan, T. M. A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions. 2006.

[4] Connor, M. The Simple, Thread-safe Approximate Nearest Neighbor (STANN) C++ Library, 2007. `http://compgeom.com/~stann/html/index.html`.

[5] Franklin, W. R. Nearest point query on 184m points in e3 with a uniform grid. In *Proceedings of the 17th Canadian Conference on Computational Geometry (CCCG'05)* (2005), pp. 239–242.

[6] Frisch, U., Matarrese, S., Mohayaee, R., and Sobolevski, A. A reconstruction of the initial conditions of the universe by optimal mass transportation. *Nature 417* (2002), 260–262.

[7] Mohayaee, R., Mathis, H., Colombi, S., and Silk, J. Reconstruction of primordial density fields. *Monthly Notices of the Royal Astronomical Society 365*, 3 (January 2006), 939–959.

[8] Mohayaee, R., and Sobolevskii, A. The Monge–Ampère–Kantorovich approach to reconstruction in cosmology. *Physica D Nonlinear Phenomena* (2008).

[9] Mohayaee, R., Tully, R. B., and Frisch, U. *Reconstruction of large-scale peculiar velocity fields*. Current Issues in Cosmology, April 2006, pp. 123–136.

[10] Mount, D. M. *ANN Programming Manual*, 2006. Available at `http://www.cs.umd.edu/~mount/ANN/`.

[11] Mount, D. M., and Arya, S. ANN: A library for approximate nearest neighbor searching, Apr 2005. `http://www.cs.umd.edu/~mount/ANN/`.

[12] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press, August 2007.

[13] Skiena, S. The Stony Brook algorithm repository, 2008. `http://www.cs.sunysb.edu/~algorith/`.