School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4040A Assignment 1 – Report

Tamlin Love (1438243) - BSc Hons

March 11, 2019

1 Introduction

The k nearest neighbour (KNN) algorithm is a simple, widely used algorithm in used for classification and regression problems[1]. Its applications vary from detecting intrusive programs[5] to text classification [4] to the analysis of nuclear magnetic resonance spectra [3].
The algorithm itself is very simple. Consider a set P of m reference points $p_i \in \mathbb{R}^d$ and a set Q of n query points $q_j \in \mathbb{R}^d$. The aim of the algorithm is to find the k nearest (according to some distance measure) points in P for each $q_j \in Q$, for some integer k.
In the brute force approach we first compute the distance between each query point and each reference point and store it in a distance matrix of size $n \times m$. We then sort each row in the matrix before returning the $n \times k$ (keeping track of any swaps via the index matrix), before returning the $n \times k$ index sub-matrix containing the indices of the k-nearest neighbours to each query point.
Special attention must be given to the distance measure and sorting functions applied above. In this report, we consider the Euclidean and Manhattan distance measures, and for sorting we consider the quicksort, bubblesort and mergesort algorithms.
Because any distance measure in $\mathbb{R}^d$ must take $\mathcal{O}(d)$ time to compute, the k nearest neighbours algorithm must take $\mathcal{O}(mnd)$ time to compute distances. Since quicksort and mergesort are both $\mathcal{O}(mlog(m))$ on average, and bubblesort is $\mathcal{O}(m^2)$ on average, the final algorithm must take $\mathcal{O}(nmlog(m))$ time to sort using quicksort or mergesort, and $\mathcal{O}(nm^2)$ time to sort using bubblesort. For large m, n and d, these complexities are problematic. However, as we shall see in Section 2, this can be significantly improved upon using parallel algorithms.

2 Methodology

As mentioned in Section 1, there are two computationally intensive sections in the algorithm: the distance computation and the sorting component. We shall apply Foster's Design Methodology to each of these sections individually in the hope of improving the performance of the algorithm.

2.1 Distance Computation

Consider the two distance measures. Because the only difference between these algorithms is the square of the difference in the Eucidean metric and the absolute value of the difference in the Manhattan metric, they can be dealt with in the same way. No matter the algorithm used, the distance function is performed nm times in the k nearest neighbours algorithm. Furthermore, the computation of the distance between points q_i and p_j is in no way dependent on the computation of distance between points q_s and p_t for $s \neq i$ and $t \neq j$. We will therefore collapse the for-loops in the algorithm which iterate over each q_i and p_j into nm tasks which can be divided among processors.
In practice, this partitioning is done by the OpenMP **for** directive with the **collapse(2)** clause. The communication, agglomeration and mapping steps are performed by the OpenMP compiler.

2.2 Sorting

Consider the sorting algorithms. These are all repeated n times, once for each q_i. Here, rather than focus on the loop over each i, we will focus on applying Foster's Design Methodology to each sorting algorithm individually.

Quicksort: Here we note that each call to Quicksort recursively calls Quicksort on two portions of the list. We can therefore partition each recursive call into a separate task to be executed by a thread. This is done using the OpenMP **sections/section** construct. However, at a certain point, the overhead involved with scheduling threads becomes significant, so that sorting a small sublist in serial is actually faster than further partitioning the task. We therefore introduce a condition: if $high - low < c$ for some cut-off c, we will sort in serial. Otherwise, we will partition further. Communication, agglomeration and mapping is then handled by the OpenMP compiler.

Mergesort: Here we note that Mergesort, like Quicksort, recursively calls Mergesort on two portions of the list. Therefore, we can parallelisze it identically: by partitioning each recursive call into a separate task using the OpenMP **sections/section** construct. Once again, we employ the cut-off condition to reduce overhead.

Bubblesort: Here we employ an algorithm known as the Odd-Even sort or the Parallel-Neighbour sort [2], which essentially sorts pairs of numbers in the list in parallel. These pairs alternate from 0-1, 2-3, 4-5, etc. (even) to 1-2, 3-4, 5-6, etc. (odd). This can be thought of as the parallel version of the bubblesort.
In this algorithm we partition each inner for-loop into a task using the OpenMP **for** directive. Once again, communication, agglomeration and mapping are handled by the OpenMP compiler.

3 Empirical Analysis

The following experiments were run on XXX with 4 cores and, therefore, 4 threads were used. Random points in $\mathbb{R}^d$ using the *generatePoints.py* file and stored in text files which were read in by the programs. For each experiment, the time taken to calculate every distance and the time taken to sort every list were measured and tabulated. Thus the total time is comprised only of these two measures.
In the first experiment, n varied between $n = 200$, $n = 800$ and $n = 1600$, with constant $d = 32$ and $m = 1000$. All 3 sorting algorithms and both distance metrics are used in the serial and parallel (using OpenMP **sections** construct) approaches. The results are tabulated below.

Figure 1: $d = 32$, $m = 1000$

			N = 200			**N = 800**			**N = 1600**	
		Dist	**Sort**	**Total**	**Dist**	**Sort**	**Total**	**Dist**	**Sort**	**Total**
Serial	**Euclid Quicksort**	0.02133	0.023122	0.044452	0.074729	1.074729	2.074729	4.074729	5.074729	6.074729
		47.98%	52.02%		3.60%	51.80%		67.08%	83.54%	
	Euclid Mergesort	0.018935	0.020924	0.039859	0.078226	0.083202	0.161428	0.146966	0.166312	0.313278
		47.50%	52.50%		48.46%	51.54%		46.91%	53.09%	
	Euclid Bubblesort	0.019371	0.531754	0.551125	0.074229	2.143703	2.217932	0.147234	4.304201	4.451435
		3.51%	96.49%		3.35%	96.65%		3.31%	96.69%	
	Manhattan Quicksort	0.048478	0.020886	0.069364	0.191908	0.084287	0.276195	0.392653	0.170683	0.563336
		69.89%	30.11%		69.48%	30.52%		69.70%	30.30%	
	Manhattan Mergesort	0.04822	0.020996	0.069216	0.191624	0.084608	0.276232	0.382254	0.166384	0.548638
		69.67%	30.33%		69.37%	30.63%		69.67%	30.33%	
	Manhattan Bubblesort	0.048939	0.532164	0.581103	0.196771	2.199062	2.395833	0.383467	4.262315	4.645782
		8.42%	91.58%		8.21%	91.79%		8.25%	91.75%	
			N = 200			**N = 800**			**N = 1600**	
		Dist	**Sort**	**Total**	**Dist**	**Sort**	**Total**	**Dist**	**Sort**	**Total**
Parallel – Sections	**Euclid Quicksort**	0.005218	0.022326	0.027544	0.019432	0.089591	0.109023	0.039463	0.178609	0.218072
		18.94%	81.06%		17.82%	82.18%		18.10%	81.90%	
	Euclid Mergesort	0.005017	0.012565	0.017582	0.02015	0.049569	0.069719	0.039265	0.097854	0.137119
		28.53%	71.47%		28.90%	71.10%		28.64%	71.36%	
	Euclid Bubblesort	0.005279	0.302504	0.307783	0.01984	1.206996	1.226836	0.039092	2.407322	2.446414
		1.72%	98.28%		1.62%	98.38%		1.60%	98.40%	
	Manhattan Quicksort	0.01245	0.022688	0.035138	0.050333	0.08931	0.139643	0.097206	0.182271	0.279477
		35.43%	64.57%		36.04%	63.96%		34.78%	65.22%	
	Manhattan Mergesort	0.012906	0.012428	0.025334	0.049283	0.048857	0.09814	0.097688	0.097834	0.195522
		50.94%	49.06%		50.22%	49.78%		49.96%	50.04%	
	Manhattan Bubblesort	0.012222	0.296567	0.308789	0.049104	1.201139	1.250243	0.097324	2.401881	2.499205
		3.96%	96.04%		3.93%	96.07%		3.89%	96.11%	

What is immediately clear is that bubblesort is by far the slowest of the sorting algorithms, in both serial and parallel. Mergesort is on average slightly faster than quicksort in serial, and noticeably faster than quicksort in parallel. Overall, the parallel approach is much faster, with quicksort being 1.77 times as fast, bubblesort being 1.85 times as fast and mergesort being 2.59 times as fast. What is also clear is that the Euclidean distance metric is, on average, faster than the Manhattan metric, in both approaches.
Consider now the same experiment with $m = 10000$.

Figure 2: $d = 32$, $m = 10000$

		N = 200			N = 800			N = 1600		
		Dist	Sort	Total	Dist	Sort	Total	Dist	Sort	Total
Serial	Euclid Quicksort	0.185437	0.278401	0.463838	0.742582	1.142975	1.885557	1.49807	2.263777	3.761847
		39.98%	60.02%		39.38%	60.62%		39.82%	60.18%	
	Euclid Mergesort	0.185513	0.270867	0.45638	0.745718	1.086814	1.832532	1.504401	2.174905	3.679306
		40.65%	59.35%		40.69%	59.31%		40.89%	59.11%	
	Euclid Bubblesort	0.184268	69.25875	69.443018	0.737184	276.673706	277.41089	1.515635	555.761849	557.277484
		0.27%	99.73%		0.27%	99.73%		0.27%	99.73%	
	Manhattan Quicksort	0.471655	0.278674	0.750329	1.932381	1.127525	3.059906	3.823178	2.227547	6.050725
		62.86%	37.14%		63.15%	36.85%		63.19%	36.81%	
	Manhattan Mergesort	0.47597	0.274112	0.750082	1.904061	1.093334	2.997395	3.903143	2.232834	6.135977
		63.46%	36.54%		63.52%	36.48%		63.61%	36.39%	
	Manhattan Bubblesort	0.471319	69.964528	70.435847	1.887251	279.456457	281.343708	3.847039	555.535158	559.382197
		0.67%	99.33%		0.67%	99.33%		0.69%	99.31%	
		N = 200			N = 800			N = 1600		
		Dist	Sort	Total	Dist	Sort	Total	Dist	Sort	Total
Parallel – Sections	Euclid Quicksort	0.047862	0.297711	0.345573	0.190373	1.194485	1.384858	0.379271	2.390482	2.769753
		13.85%	86.15%		13.75%	86.25%		13.69%	86.31%	
	Euclid Mergesort	0.072815	0.155163	0.227978	0.190162	0.625722	0.815884	0.388684	1.25202	1.640704
		31.94%	68.06%		23.31%	76.69%		23.69%	76.31%	
	Euclid Bubblesort	0.047736	16.933148	16.980884	0.19027	66.511185	66.701455	0.383545	142.115242	142.498787
		0.28%	99.72%		0.29%	99.71%		0.27%	99.73%	
	Manhattan Quicksort	0.120505	0.297456	0.417961	0.480964	1.194215	1.675179	0.974073	2.382374	3.356447
		28.83%	71.17%		28.71%	71.29%		29.02%	70.98%	
	Manhattan Mergesort	0.12055	0.154432	0.274982	0.481281	0.626853	1.108134	0.966649	1.241424	2.208073
		43.84%	56.16%		43.43%	56.57%		43.78%	56.22%	
	Manhattan Bubblesort	0.120661	16.675757	16.796418	0.481504	68.690477	69.171981	1.251648	132.479225	133.730873
		0.72%	99.28%		0.70%	99.30%		0.94%	99.06%	

Here it is even more clear that the bubblesort is inefficient, as its time complexity scales quadratically in m. With such a large m, the advantages of the parallel approach are even more apparent, with quicksort speeding up by a factor of 1.61, mergesort by 2.53 and bubblesort by 4.07.
We now vary $d = 64, 128, 256, 512$ and keep $n = 800$, $m = 5000$ constant. For brevity's sake, we consider only the quicksort algorithm.

Figure 3: $n = 800$, $m = 5000$

		N = 800, M = 5000											
		D = 64			D = 128			D = 256			D = 512		
		Dist	Sort	Total	Dist	Sort	Total	Dist	Sort	Total	Dist	Sort	Total
Serial	Euclid Quicksort	0.674948	0.515213	1.190161	1.315786	0.515559	1.831345	2.841606	0.535521	3.377127	5.132581	0.506111	5.638692
		56.71%	43.29%		71.85%	28.15%		84.14%	15.86%		91.02%	8.98%	
	Manhattan Quicksort	1.976354	0.520283	2.496637	4.081035	0.512601	4.593636	8.34295	0.510896	8.853846	16.495382	0.506719	17.002101
		79.16%	20.84%		88.84%	11.16%		94.23%	5.77%		97.02%	2.98%	
		D = 64			D = 128			D = 256			D = 512		
		Dist	Sort	Total	Dist	Sort	Total	Dist	Sort	Total	Dist	Sort	Total
Parallel – Sections	Euclid Quicksort	0.177186	0.553123	0.730309	0.351709	0.55312	0.904829	0.686863	0.550247	1.23711	1.34542	0.541005	1.886425
		24.26%	75.74%		38.87%	61.13%		55.52%	44.48%		71.32%	28.68%	
	Manhattan Quicksort	0.501356	0.551249	1.052605	1.285376	0.600948	1.886324	2.097209	0.54858	2.645789	4.200083	0.543786	4.743869
		47.63%	52.37%		68.14%	31.86%		79.27%	20.73%		88.54%	11.46%	

Here it is easily observed that the Manhattan metric is much slower than the Euclidean one. This is likely due to the presence of a conditional statement when evaluating $|p_i - q_i|$. We also note the effectiveness of the parallelisation, which on average sped the Euclidean calculation by a factor of 2.74 and the Manhattan calculation by a factor of 3.19.

4 Summary

References

[1] N. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[2] A. N. Habermann, “Parallel neighbor-sort (or the glory of the induction principle),” 1972.

[3] B. R. Kowalski and C. F. Bender, “K-nearest neighbor classification rule (pattern recognition) applied to nuclear magnetic resonance spectral interpretation,” *Analytical Chemistry*, 1972.

[4] O.-W. Kwon and J.-H. Lee, “Text categorization based on k-nearest neighbor approach for web site classification,” *Information Processing and Management*, 2003.

[5] Y. Liao and V. R. Vemuri, “Use of k-nearest neighbor classifier for intrusion detection,” *Computers and Security*, 2002.