PROGRAMMING ASSIGNMENT 1

Matthew Michaelson and Rodney Lafuente Mercado

NOTE: Matthew is an extension student and so as per discussion with Eric Knorr, we are turning this in by the extension student deadline. In addition we are each turning in a copy, also per Eric's instructions.

Summary

We decided to do our assignment in C to take advantage of the simplicity and low-level of the language. We then decided to try both Kruskal's and Prim's algorithm for this assignment since this way we could compare the two results and see if they were the same as a testing aid. In terms of speed, the versions presented in class were asymptotically comparable, with Prim's at O(|E| log |V|) (where the logarithm is of base |E|/|V|, since Prim's is implemented with a d-ary heap) and Kruskal's at O(|E| log |V|) if the edges are not already sorted (though it would be possible to improve Kruskal's if we handled sorting better--see below in the discussion of Kruskal's). In the end we decided that our Prim's algorithm ran faster and so we include only those results, but the code for Kruskal's will be included as an appendix.

For each table of results we ran Prim's algorithm five times on each given n and averaged to calculate the average column.

Tables of Results

	0	2	3	4
128	1.8265	6.705201	16.09189	26.164528
256	1.258418	9.416765	25.845562	45.572159
512	1.209792	13.076204	40.190784	75.407433
1024	1.184928	18.127352	63.62915	122.81591
2048	1.213897	25.67104	100.276596	209.513321
4096	1.186041	36.451134	157.540359	345.525055
8192	1.202036	51.596191	250.602631	580.196533
16384	1.202978	72.826797	397.492249	970.147766
32768	1.202568	103.238785	628.486023	1627.942139
65536	1.203485	6210.287598	13025.25781	2725.722989
131072	1.203506	12610.43359	26650.70131	4910.273823
262144	1.200969			

Estimate for f(n): It appears that in the dimension 0 trials the average seems to be asymptotically approaching a certain constant as n increases. We can see that with small n the variance from run to run of the algorithm is larger, and as n increases the variance goes down and the average seems to hover closer and closer to some number around 1.2. After doing some research we believe that the constant it seems to approach is ZETA(3) i.e. the Riemann Zeta function evaluated at input 3, otherwise apparently known as the Apery Constant.

With dimension = 1, (not part of this assignment but still relevant), the weights of the MST also converged to a constant. This constant turned out to be half the Riemann Zeta function of 3, which concurs with a theorem developed by mathematician Alan M. Frieze that states that the weights of random minimum spanning trees on complete graphs converge to ZETA(3)/D where D is the derivative of the cumulative distribution function of the weights at 0. It turns out that the distance between uniformly distributed points on a line segment between 0 and 1 has a triangular distribution whose CDF derivative at 0 is ½.

As for higher dimensions, the weights of the trees no longer converged to a constant. The function f(n) is exponential, and the best constant we came up with for the base was approximately 1.4. In other words, our estimate is f(n) = 1.4^n

A very notable issue we ran into was overflowing our stack on higher values of numpoints when we didn't take away enough edges. The problem with taking more edges was that the algorithm no longer output correct results. This led us to use a special case of k(n) for dimension = 4 than we did for other dimensions.

Discussion

Our process was to first aim for accuracy, and then improve efficiency. In general we found that it was helpful to have more than one algorithm to compare against when ensuring our results were correct — even though in the end we went with Prim's this process helped us to find errors in both algorithms when we noticed that they weren't matching up and went looking for an explanation in each one, we found errors in both. In order to ensure our algorithms were accurate, we tested them on small inputs and compared the results step by step by hand with manual calculations. For higher dimensions and for larger n's this was not practical, but we developed expectations based on the small n's and also were able to compare the two algorithms against each other.

In setting up our random number generator we experimented with different seeding setups and finally settled on a system time-based seeding that produces pseudorandom numbers with close to uniform variability. We did some research to find out about Mersenne Twisters, since we felt our initial approach could be improved, and if given more time we would switch to an implementation of a Mersenne Twister for even better pseudo-randomness.

After we ensured the two were accurate we focused on improving efficiency. Here we found the concept of lazy memory allocation and variable initialization helpful, and the suggestion in the

hint very helpful. We tried to determine k(n) by running the algorithms on different inputs and observing the following: the average edge weight, maximum edge weight of an edge actually used in the MST, and then comparing those to the MST total weight at different values of n. This empirical approach suggested k(n). We tested that our k(n) was not removing too many edges by running both algorithms with and without it to compare results, and by pulling out the maximum edge of our MST with and without different k functions to see if it was affected.

Our experience with Prim's in particular was that a K function of K(n) = 1/sqrt(n) was not taking too many edges for dimension 3 and below, however, for dimension = 4 it altered the results of the algorithm significantly.

Description of Experience with Kruskals:
Though we didn't end up using it, our experience with Kruskal's in particular was that it was much more difficult to correctly implement disjoint sets than it was to implement Kruskal's algorithm itself (which was straightforward). Then however, we discovered that the initial naive approach of just creating the entire graph in the beginning was prohibitively memory-expensive, so we changed our approach to allocate memory for and initialize each needed edge as lazily as possible i.e. right in a loop that determined how many edges we'd make in the first place. Then when we added a cutoff based on a k(n) function described above, we could simply skip the creation of unneeded edges entirely in that initial loop. Another optimization that we ran out of time to perform here would be to sort as we go, instead of performing quicksort at the end of edge creation. This would theoretically allow us to get the O(|E| Ackermann(V)) time discussed in class.

In the end in terms of time Kruskal's took less than 1 minute for a single iteration on up to n=65536, but after that point it took so long that we weren't able to run it on one of our laptops. This is interesting because when we looked closer we could see that the increase in time as n increased was something like 4 times for each doubling of n, but at n=131072, suggesting that something other than the algorithm was responsible for the long operation times at high n. In fact we believe it has to do with memory limitations on the laptop we were running on -- for instance once the total array of edges to be sorted in the sorting step can no longer all fix in cache, the need to hit the hard disk will slow down the program considerably. In the end however, this was enough to push us to focus on Prim's.

APPENDIX: Kruskal Implementation

```
/* KRUSKALs
// takes a graph g(v, e), returns an MST x
// handles original graph data structures
// params:
//  0 -- just return x
//  1 -- return total weight of MST
//  2 -- return total weight of MST and path
//  3 -- show time it takes for different parts
//  4 -- show stats and time elapsed
//  5 -- show all edges as made */

djset* kruskal(Graph_nd* graph, int param, int dimension) {
    /* sort all edges smallest-to-largest
        greedily add smaller edges unless adding an edge creates a cycle
        stop at |V|-1 edges  */
    time_t time1;
    time(&time1);

    if (param == 4) {
        printf("Running...\n");
    }

    if (param == 3) {
        print_time("KRUSKAL - Beginning (allocating djsets)");
    }

    // initialize arrays for djsets
    djset** sets = (djset**) malloc(graph->V * sizeof(djset));
    float path_weight = 0;

    if (param == 3) {
        print_time("Before sorting");
    }

    // initialize empty set x (our MST)
    djset* x = dj_makeset(-1);

    // calculate edge weights based on graph dimension
    Edge** e;
    e = get_sort_graph_nd_weights(graph);
```

```
if (param == 2 || param == 5) {
    for (int i = 0; i < graph->cutoff_m; i++) {
        printf("(%d, %d), weight: %f ", e[i]->u, e[i]->v, e[i]->weight);
        printf("d\n");
    }
}

if (param == 3) {
    print_time("After sorting, before makesets");
}

// make each vertex into a set including just itself
for (int i=0; i < graph->V; i++) {
     sets[i] = dj_makeset(i);
}

if (param == 3) {
    print_time("Beginning of loop");
}

/* for all edges in the graph, if they're not
    in the same disjoint set, then add them to our MST x,
    and union them to each other */
float max_edge_weight = 0.0;
for (int i=0; i < graph->cutoff_m; i++) {
        if (param == 5) {
            printf("OUTER\n");
            printf("u find: %p\n", dj_find(sets[(e[i]->u - 1)]));
            printf("v find: %p\n", dj_find(sets[(e[i]->v - 1)]));

            printf("u: ");
            print_djset(sets[(e[i]->u - 1)], 1);
            printf("v: ");
            print_djset(sets[(e[i]->v - 1)], 1);
        }

    if (dj_find(sets[(e[i]->u - 1)]) != dj_find(sets[(e[i]->v - 1)])) {

        // print actual MST if param chosen
        if (param == 2 || param == 5) {
            if (param == 5) {
                printf("INNER: \n");
```

```
                printf("u find: %p\n", dj_find(sets[(e[i]->u - 1)]));
                printf("v find: %p\n", dj_find(sets[(e[i]->v - 1)]));
            }

            printf("u: ");
            print_djset(sets[(e[i]->u - 1)], 1);
            printf("v: ");
            print_djset(sets[(e[i]->v - 1)], 1);
        }

        // key step: actually union things into your MST
        //dj_union(sets[(e[i]->v - 1)], x);
        dj_union(sets[(e[i]->u - 1)], sets[(e[i]->v - 1)]);

        // add up the weight of the MST if proper param chosen
        if (param <=  2 || param == 4 || param == 5) {
            path_weight += e[i]->weight;
            max_edge_weight = e[i]->weight;
        }
      }
   }
   if (param <=  2 || param == 4 || param == 5) {
      printf("MST path weight: %f\n", path_weight);
      printf("Average edge weight: %f\n", (path_weight/graph->cutoff_m));
      printf("Max edge weight: %f\n\n", max_edge_weight);
   }

   if (param == 3 ) {
      print_time("Ending - ");
   }
   if (param == 4) {
      time_t time2;
      time(&time2);
      printf("End. Elapsed time is  %.2f seconds\n",
      difftime(time2, time1));
   }

   return x;
}

djset* dj_makeset(int x) {
        djset* new_djset = (djset*) malloc(sizeof(djset));
        new_djset->parent = new_djset;
```

```
	new_djset->value = x;
	new_djset->rank = 0;
	return new_djset;
}

djset* dj_find(djset* x) {
	if (x != x->parent) {
		x->parent = dj_find(x->parent);
	}
	return x->parent;
}

void dj_link(djset* x, djset* y) {
	if (x->rank > y->rank) {
		// switch x and y
		djset* temp = y;
		y = x;
		x = temp;
	}
	if (x->rank == y->rank) {
		y->rank++;
	}
	//printf("x->parent: %p\n", x->parent);
	//printf("y: %p\n", y);
	x->parent = y;
	//printf("x->parent: %p\n", x->parent);
}

void dj_union(djset* x, djset* y) {
	dj_link(dj_find(x), dj_find(y));
}

int cmpfunc_edge (const void* a, const void* b) {
   Edge* ea = *(Edge**) a;
   Edge* eb = *(Edge**) b;
   return (ea->weight > eb->weight) - (ea->weight < eb->weight);
}

// lazy heapsorting graph edge weights (for kruskal)
Edge** get_sort_graph_nd_weights(Graph_nd* graph) {
   //float* v_weights = (float*) malloc(graph->V * sizeof(float));
   //EdgeHeap* e_values = createEdgeHeap(graph->m, -1); // TODO shrink this!
   long long cutoff_m = graph->cutoff_m; //
```

```
int dim = graph->dim;
float c = 1.0;//((float)graph->V/(float)graph->m) * 50.0* dim;
float cutoff_w = c;
//printf("dim: %d\n", dim);
//printf("total edges: %lli\n", cutoff_m);
//Edge** edges= (Edge**) malloc((cutoff_m) * sizeof(Edge*));
Array edges;
initArray(&edges, 1);

// for each edge, calculate edge weight then heapify if small enough
// fill in edge vertices
int k = 0;
int edges_made = 0;
while (k < cutoff_m) {
    for (int i = 0; i < graph->V; i++) {
        for (int j = 0; j < graph->V; j++) {
            if (i < j && k < cutoff_m) {
                // calculate weight in d dimensions:
                //printf("graph dim: %d\n", graph->dim);
                float weight = 0.0;
                if (dim ==0) {
                    weight = randf();
                } else {
                    for (int d = 0; d < dim; d++) {
                        float r1 = randf();
                        float r2 = randf();
                        float r = r1-r2;
                        weight += r*r;
                        //printf("r1: %f, r2: %f, diff: %f, times: %f\n", r1, r2, r, weight);
                    }
                    weight = sqrtf(weight);
                }

                // if weight is small enough, then add in to edges
                if (weight < cutoff_w) {
                    Edge* new_edge = (Edge*)malloc(sizeof(Edge));
                    edges_made++;
                    new_edge->u = i+1;
                    new_edge->v = j+1;
                    new_edge->weight = weight;
                    insertArray(&edges, new_edge);
                }
                k++;
```

```
            }
        }
    }
}

graph->cutoff_m = edges_made;

//printf("edges made: %d\n", edges_made);

qsort(edges.array, edges_made, sizeof(Edge*), cmpfunc_edge);


/*
// testing sort
for (int i = 0; i < edges_made; i++) {
    printf("Sorted pointer %d: %p, with weight: %f\n",
        i,
        edges.array[i],
        edges.array[i]->weight);    }
*/

return edges.array;
}

// Function to print an array
void printarray(float arr[], int size) {
    int i;
    for (i=0; i < size; i++)
        printf("%f ", arr[i]);
    printf("|| size: %d\n", size);
}

// array data structure -- for Edges!
void initArray(Array *a, size_t initialSize) {
  a->array = (Edge **)malloc(initialSize * sizeof(Edge*));
  a->used = 0;
  a->size = initialSize;
}

void insertArray(Array *a, Edge* element) {
  // a->used is the number of used entries, because a->array[a->used++] updates a->used only
*after* the array has been accessed.
  // Therefore a->used can go up to a->size
```

```
  if (a->used == a->size) {
    a->size*=2; //++; //*= 2;
    a->array = (Edge **)realloc(a->array, a->size * sizeof(Edge*));
  }
  a->array[a->used++] = element;
}

void freeArray(Array *a) {
  free(a->array);
  a->array = NULL;
  a->used = a->size = 0;
}

typedef struct Array {
  Edge** array;
  size_t used;
  size_t size;
} Array;

typedef struct {
        float weight;
        int u;
        int v;
} Edge;
```