Li Yicheng

2014  4P3

2015 noi selection test

emergency code reference

(for common problems)

# Coin Change Problem

Given a sum of money e.g. 75 cents and denominations e.g. 5 cents, 10 cents, and 50 cents, find the least number of coins required to make the change.



# Dijkstra’s Algorithm – Shortest Path Problem

To find the shortest path in a graph.

#include <iostream>

#include <vector>

#include <string>

#include <list>

#include <limits> // for numeric\_limits

#include <set>

#include <utility> // for pair

#include <algorithm>

#include <iterator>

typedef int vertex\_t;

typedef double weight\_t;

const weight\_t max\_weight = std::numeric\_limits<double>::infinity();

struct neighbor {

vertex\_t target;

weight\_t weight;

neighbor(vertex\_t arg\_target, weight\_t arg\_weight)

: target(arg\_target), weight(arg\_weight) { }

};

typedef std::vector<std::vector<neighbor> > adjacency\_list\_t;

void DijkstraComputePaths(vertex\_t source,

const adjacency\_list\_t &adjacency\_list,

std::vector<weight\_t> &min\_distance,

std::vector<vertex\_t> &previous)

{

int n = adjacency\_list.size();

min\_distance.clear();

min\_distance.resize(n, max\_weight);

min\_distance[source] = 0;

previous.clear();

previous.resize(n, -1);

std::set<std::pair<weight\_t, vertex\_t> > vertex\_queue;

vertex\_queue.insert(std::make\_pair(min\_distance[source], source));

while (!vertex\_queue.empty())

{

weight\_t dist = vertex\_queue.begin()->first;

vertex\_t u = vertex\_queue.begin()->second;

vertex\_queue.erase(vertex\_queue.begin());

// Visit each edge exiting u

const std::vector<neighbor> &neighbors = adjacency\_list[u];

for (std::vector<neighbor>::const\_iterator neighbor\_iter = neighbors.begin();

neighbor\_iter != neighbors.end();

neighbor\_iter++)

{

vertex\_t v = neighbor\_iter->target;

weight\_t weight = neighbor\_iter->weight;

weight\_t distance\_through\_u = dist + weight;

if (distance\_through\_u < min\_distance[v]) {

vertex\_queue.erase(std::make\_pair(min\_distance[v], v));

min\_distance[v] = distance\_through\_u;

previous[v] = u;

vertex\_queue.insert(std::make\_pair(min\_distance[v], v));

}

}

}

}

std::list<vertex\_t> DijkstraGetShortestPathTo(

vertex\_t vertex, const std::vector<vertex\_t> &previous)

{

std::list<vertex\_t> path;

for ( ; vertex != -1; vertex = previous[vertex])

path.push\_front(vertex);

return path;

}

int main()

{

// remember to insert edges both ways for an undirected graph

adjacency\_list\_t adjacency\_list(6);

// 0 = a

adjacency\_list[0].push\_back(neighbor(1, 7));

adjacency\_list[0].push\_back(neighbor(2, 9));

adjacency\_list[0].push\_back(neighbor(5, 14));

// 1 = b

adjacency\_list[1].push\_back(neighbor(0, 7));

adjacency\_list[1].push\_back(neighbor(2, 10));

adjacency\_list[1].push\_back(neighbor(3, 15));

// 2 = c

adjacency\_list[2].push\_back(neighbor(0, 9));

adjacency\_list[2].push\_back(neighbor(1, 10));

adjacency\_list[2].push\_back(neighbor(3, 11));

adjacency\_list[2].push\_back(neighbor(5, 2));

// 3 = d

adjacency\_list[3].push\_back(neighbor(1, 15));

adjacency\_list[3].push\_back(neighbor(2, 11));

adjacency\_list[3].push\_back(neighbor(4, 6));

// 4 = e

adjacency\_list[4].push\_back(neighbor(3, 6));

adjacency\_list[4].push\_back(neighbor(5, 9));

// 5 = f

adjacency\_list[5].push\_back(neighbor(0, 14));

adjacency\_list[5].push\_back(neighbor(2, 2));

adjacency\_list[5].push\_back(neighbor(4, 9));

std::vector<weight\_t> min\_distance;

std::vector<vertex\_t> previous;

DijkstraComputePaths(0, adjacency\_list, min\_distance, previous);

std::cout << "Distance from 0 to 4: " << min\_distance[4] << std::endl;

std::list<vertex\_t> path = DijkstraGetShortestPathTo(4, previous);

std::cout << "Path : ";

std::copy(path.begin(), path.end(), std::ostream\_iterator<vertex\_t>(std::cout, " "));

std::cout << std::endl;

return 0;

}

# Palindrome Detection

To detect if a given string is a palindrome. A palindrome is something where that is exactly the same reversed e.g. americacirema

#include <string>

#include <algorithm>

bool is\_palindrome(std::string const& s)

{

return std::equal(s.begin(), s.begin()+s.length()/2, s.rbegin());

}

# Towers of Hanoi Solution

A bot to play the Towers of Hanoi game. It involves shifting circular blocks of varying sizes from one pole to another, with another empty pole to help. A block can sit on a larger block on the same pole, but a larger block cannot sit on a smaller block. Only one block can be shifted at the same time.

void move(int n, int from, int to, int via) {

if (n == 1) {

std::cout << "Move disk from pole " << from << " to pole " << to << std::endl;

} else {

move(n - 1, from, via, to);

move(1, from, to, via);

move(n - 1, via, to, from);

}

}

# Closest Pair Problem

Finds the 2 points on a 2D plane that are closest.

#include <stdio.h>

#include <stdlib.h>

#include <values.h>

#include <math.h>

#include <string.h>

typedef struct { double x, y; } point\_t, \*point;

inline double dist(point a, point b)

{

double dx = a->x - b->x, dy = a->y - b->y;

return dx \* dx + dy \* dy;

}

inline int cmp\_dbl(double a, double b)

{

return a < b ? -1 : a > b ? 1 : 0;

}

int cmp\_x(const void \*a, const void \*b) {

return cmp\_dbl( (\*(const point\*)a)->x, (\*(const point\*)b)->x );

}

int cmp\_y(const void \*a, const void \*b) {

return cmp\_dbl( (\*(const point\*)a)->y, (\*(const point\*)b)->y );

}

double closest(point\* sx, int nx, point\* sy, int ny, point \*a, point \*b)

{

int left, right, i;

double d, min\_d, x0, x1, mid, x;

point a1, b1;

point \*s\_yy;

if (nx <= 8) return brute\_force(sx, nx, a, b);

s\_yy = malloc(sizeof(point) \* ny);

mid = sx[nx/2]->x;

/\* adding points to the y-sorted list; if a point's x is less than mid,

add to the begining; if more, add to the end backwards, hence the

need to reverse it \*/

left = -1; right = ny;

for (i = 0; i < ny; i++)

if (sy[i]->x < mid) s\_yy[++left] = sy[i];

else s\_yy[--right]= sy[i];

/\* reverse the higher part of the list \*/

for (i = ny - 1; right < i; right ++, i--) {

a1 = s\_yy[right]; s\_yy[right] = s\_yy[i]; s\_yy[i] = a1;

}

min\_d = closest(sx, nx/2, s\_yy, left + 1, a, b);

d = closest(sx + nx/2, nx - nx/2, s\_yy + left + 1, ny - left - 1, &a1, &b1);

if (d < min\_d) { min\_d = d; \*a = a1; \*b = b1; }

d = sqrt(min\_d);

/\* get all the points within distance d of the center line \*/

left = -1; right = ny;

for (i = 0; i < ny; i++) {

x = sy[i]->x - mid;

if (x <= -d || x >= d) continue;

if (x < 0) s\_yy[++left] = sy[i];

else s\_yy[--right] = sy[i];

}

/\* compare each left point to right point \*/

while (left >= 0) {

x0 = s\_yy[left]->y + d;

while (right < ny && s\_yy[right]->y > x0) right ++;

if (right >= ny) break;

x1 = s\_yy[left]->y - d;

for (i = right; i < ny && s\_yy[i]->y > x1; i++)

if ((x = dist(s\_yy[left], s\_yy[i])) < min\_d) {

min\_d = x;

d = sqrt(min\_d);

\*a = s\_yy[left];

\*b = s\_yy[i];

}

left --;

}

free(s\_yy);

return min\_d;

}

#define NP 1000000

int main()

{

int i;

point a, b;

point pts = malloc(sizeof(point\_t) \* NP);

point\* s\_x = malloc(sizeof(point) \* NP);

point\* s\_y = malloc(sizeof(point) \* NP);

for(i = 0; i < NP; i++) {

s\_x[i] = pts + i;

pts[i].x = 100 \* (double) rand()/RAND\_MAX;

pts[i].y = 100 \* (double) rand()/RAND\_MAX;

}

/\* printf("brute force: %g, ", sqrt(brute\_force(s\_x, NP, &a, &b)));

printf("between (%f,%f) and (%f,%f)\n", a->x, a->y, b->x, b->y); \*/

memcpy(s\_y, s\_x, sizeof(point) \* NP);

qsort(s\_x, NP, sizeof(point), cmp\_x);

qsort(s\_y, NP, sizeof(point), cmp\_y);

printf("min: %g; ", sqrt(closest(s\_x, NP, s\_y, NP, &a, &b)));

printf("point (%f,%f) and (%f,%f)\n", a->x, a->y, b->x, b->y);

/\* not freeing the memory, let OS deal with it. Habit. \*/

return 0;

}

# Dutch National Flag Problem

Given a number of red, blue and white balls in random order, arrange them in the order of the colours in the Dutch national flag.

#include <algorithm>

#include <iostream>

// Dutch national flag problem

template <typename BidIt, typename T>

void dnf\_partition(BidIt first, BidIt last, const T& low, const T& high)

{

for (BidIt next = first; next != last; ) {

if (\*next < low) {

std::iter\_swap(first++, next++);

} else if (!(\*next < high)) {

std::iter\_swap(next, --last);

} else {

++next;

}

}

}

enum Colors { RED, WHITE, BLUE };

void print(const Colors \*balls, size\_t size)

{

static const char \*label[] = { "red", "white", "blue" };

std::cout << "Balls:";

for (size\_t i = 0; i < size; ++i) {

std::cout << ' ' << label[balls[i]];

}

std::cout << "\nSorted: " << std::boolalpha << std::is\_sorted(balls, balls + size) << '\n';

}

int main()

{

Colors balls[] = { RED, WHITE, BLUE, RED, WHITE, BLUE, RED, WHITE, BLUE };

std::random\_shuffle(balls, balls + 9);

print(balls, 9);

dnf\_partition(balls, balls + 9, WHITE, BLUE);

print(balls, 9);

}

# Maximum Subarray Problem

Given an array e.g. [1,2,-3,2] find the subarray with the maximum sum ([1,2], sum is 3).

int maxSubArray(int[] A) {

int newsum=A[0];

int max=A[0];

for(int i=1;i<A.length;i++){

newsum=Math.max(newsum+A[i],A[i]);

max= Math.max(max, newsum);

}

return max;

}