

```
===== search.h =====
```

```
#ifndef _SEARCH_H_
#define _SEARCH_H_

#include <iostream>
#include <vector>

class state {
public:
    state();
    virtual ~state();

    // callee owns the memory
    virtual std::vector<state *> succ() const = 0;
    // current cost:
    virtual double pathlength() const = 0;
    // heuristic function:
    virtual double mincosttogo() const = 0;
    // is this the goal?
    virtual bool endstate() const = 0;
    // used for debugging
    virtual void display(std::ostream &os) const {}
    // used to clean up information related to path if necessary
    virtual void clean() {}

    virtual bool equal(const state *s) const { return false; }
};

state *search(state *start);

#endif
```

```
===== search.cpp =====
```

```
#include "search.h"
#include <queue>

// comment out next line to remove closed list
#define CLOSEDLIST

using namespace std;

state::state() {}

state::~state() {}

class searchstate {
public: // all public b/c it is local to this .cc file
    searchstate(state *ss) {
        s = ss;
        r = -s->pathlength()-s->mincosttogo();
    }
    // does not delete s!! (it doesn't really belong to us!)
    // this is a little strange and must be remembered!
    virtual ~searchstate() {}

    // These are for priority_queue comparisons (why not have all 4?)
    bool operator>(const searchstate &ss) const { return r>ss.r; }
    bool operator>=(const searchstate &ss) const { return r>=ss.r; }
    bool operator<(const searchstate &ss) const { return r<ss.r; }
    bool operator<=(const searchstate &ss) const { return r<=ss.r; }

    state *s;
    double r; // this is kept (cached) here for quick evaluation
};

// This closed *list* should be replaced with a hash table
// However, STL doesn't provide a true hash table and the syntax
// gets a little more ugly. The comparison operators above are
// for the pqueue and you need a different set for the map (or hash table).
// Therefore, you need to pass in comparison classes. To keep things
// clean and simple, I have dropped this. However, you should feel
// free to implement it.
static bool ismember(const state *ss, vector<state *> cl) {
    for(int i=0;i<cl.size();i++)
        if (ss->equal(cl[i])) return true;
    return false;
}
```

```

// This function is long for two reasons:
// 1. I'm making sure I clean up all of the memory
// 2. I have all of the non-closed list code and the closed list code
// Really, it isn't that long.
state *search(state *start) {
    priority_queue<searchstate> openlist;
    openlist.push(searchstate(start));
#ifdef CLOSEDLIST
    vector<state *> closelist;
    closelist.push_back(start);
#endif

    while(!openlist.empty()) {
        searchstate curr = openlist.top();
        openlist.pop();
        if (curr.s->endstate()) { // found the answer!
            // we need to clean up...
#ifdef CLOSEDLIST
            while(!openlist.empty()) {
                searchstate todel = openlist.top();
                openlist.pop();
                if (todel.s!=start && todel.s!=curr.s)
                    delete todel.s;
            }
#else
            for(int i=0;i<closelist.size();i++)
                if (closelist[i] != start && closelist[i]!=curr.s)
                    delete closelist[i];
#endif
            return curr.s;
        }

        // add successors...
        vector<state *> toadd = curr.s->succ();
        for(int i=0;i<toadd.size();i++) {
#ifdef CLOSEDLIST
            if (ismember(toadd[i],closelist)) {
                delete toadd[i];
                continue;
            }
            closelist.push_back(toadd[i]);
#endif
            openlist.push(searchstate(toadd[i]));
#ifdef CLOSEDLIST
            if (curr.s != start) delete curr.s; // clean space
#else
            curr.s->clean(); // let it know that it is just on the
                // closed list and therefore can remove information
#endif
        }
#ifdef CLOSEDLIST
        for(int i=0;i<closelist.size();i++)
            if (closelist[i] != start) delete closelist[i];
#endif
    }
    return NULL; // could not find any solution
}

```

```

===== mazestate.h =====
#ifndef _MAZESTATE_H_
#define _MAZESTATE_H_

#include "maze.h"
#include "search.h"
#include "pose.h"

class mazestate : public state {
public:
    mazestate(const maze *mm, int goalx, int goaly, const std::vector<pose> &start
);
    virtual ~mazestate();

    virtual std::vector<state *> succ() const;
    virtual double pathlength() const;
    virtual double mincosttogo() const;
    virtual bool endstate() const;

    virtual void display(std::ostream &os) const;

    virtual bool equal(const state *s) const;
    virtual void clean();

    std::vector<int> path() { return moves; }

private:
    int gx,gy;
    const maze *m;
    int nstep;
    std::vector<int> moves;
    std::vector<pose> loc;
};
#endif

```

```

===== mazestate.cpp =====
#include "mazestate.h"

using namespace std;

mazestate::mazestate(const maze *mm, int goalx, int goaly, const std::vector<pose> &st
art) : m(mm), loc(start) {
    gx = goalx;
    gy = goaly;
    nstep = 0;
}

mazestate::~mazestate() {
}

static void setinsert(pose p, vector<pose> &s) {
    for(int i=0;i<s.size();i++)
        if (s[i].location==p.location
            && s[i].direction==p.direction) return;
    s.push_back(p);
}

vector<state *> mazestate::succ() const {
    vector<state *> ret;
    for(int movenum=0;movenum<3;movenum++) {
        mazestate *next = new mazestate(*this);
        next->moves.push_back(movenum);
        next->loc.clear();
        for(int i=0;i<loc.size();i++) {
            pose p = loc[i];
            switch(movenum) {
                case 0: {
                    int x,y;
                    m->indexposition(p.location,x,y);
                    bool w[4];
                    m->localwalls(x,y,w);
                    if (!w[p.direction]) {
                        m->move(x,y,p.direction);
                        p.location = m->positionindex(x,y);
                    }
                }
                case 1: p.direction = (p.direction+1)%4;
                        break;
                case 2: p.direction = (p.direction+3)%4;
                        break;
            }
            setinsert(p,next->loc);
        }
        next->nstep++;
        ret.push_back(next);
    }
    return ret;
}

double mazestate::pathlength() const {
    return nstep;
}

double mazestate::mincosttogo() const {
    int ret = 0;
    for(int i=0;i<loc.size();i++) {
        int x,y;
        m->indexposition(loc[i].location,x,y);
        int mval = abs(x-gx)+abs(y-gy);
        if (mval<ret) ret = mval;
    }
    return ret;
}

```

```

bool mazestate::endstate() const {
    int gi = m->positionindex(gx,gy);
    for(int i=0;i<loc.size();i++)
        if (gi!=loc[i].location) return false;
    return true;
}

void mazestate::display(ostream &os) const {
    vector<int> l;
    for(int i=0;i<loc.size();i++)
        l.push_back(loc[i].location);
    m->display(os,l);
}

bool mazestate::equal(const state *s) const {
    const mazestate *ss = dynamic_cast<const mazestate *>(s);
    if (ss==NULL) return false;
    if (loc.size() != ss->loc.size()) return false;
    for(int i=0;i<loc.size();i++) {
        int j;
        for(j=0;j<ss->loc.size();j++)
            if (loc[i].location == ss->loc[j].location
                && loc[i].direction == ss->loc[j].direction)
                break;
        if (j==ss->loc.size()) return false;
    }
    return true;
}

void mazestate::clean() {
    moves.clear();
}

===== main.cpp [the important part] =====
...
#include "mazestate.h"
...
vector<int> findmoves(const maze &m, int goalx, int goaly, const vector<pose> &startpo
ses, bool &found) {
    mazestate *start = new mazestate(&m,goalx,goaly,startposes);
    mazestate *sol = dynamic_cast<mazestate *>(search(start));
    found = (sol!=NULL);
    if (sol == NULL) return vector<int>();
    vector<int> retpath = sol->path();
    delete start;
    delete sol;
    return retpath;
}

```