A* Graph Search Within the BGL Framework

Kris Beevers
beevek@cs.rpi.edu

Jufeng Peng
pengj@cs.rpi.edu

December 4, 2003

Abstract

We introduce an implementation of the A* heuristic graph search algorithm that works within the Boost Graph Library (BGL) framework. BGL is a highly generic, C++ template-based graph library, created with the aim of providing efficient containers and algorithms that can be easily reused in different applications. Our A* implementation is written with these goals in mind. We also present usage examples and a reference manual, along with results of correctness and performance tests of our implementation.

Contents

List of Algorithms

1 User's Guide

1.1 Introduction to A*

The A* algorithm is a *heuristic graph search algorithm*: an A* search is "guided" by a *heuristic function*. A heuristic function $h(v)$ is one which estimates the cost from a non-goal state (v) in the graph to some goal state, g. Intuitively, A* follows paths (through the graph) to the goal that are estimated by the heuristic function to be the best paths. Unlike best-first search, A* takes into account the known cost from the start of the search to v; the paths A* takes are guided by a function

$$f(v) = g(v) + h(v)$$

where $h(v)$ is the heuristic function, and $g(v)$ (sometimes denoted $c(s, v)$) is the known cost from the start to v. Clearly, the efficiency of A* is highly dependent on the heuristic function with which it is used.

Heuristic functions can take on several properties. An *admissible* heuristic is one which never overestimates the cost to the goal (i.e. its estimate is always less than or equal to the true cost, $c(v, g)$). A *consistent* or *monotonic* heuristic is one for which, for any two vertices v_1 and v_2, $h(v_2) \geq h(v_1) + c(v_1, v_2)$. (Note that this is just the triangle inequality.)

Given an admissible and consistent heuristic, A* provably examines, and can return, the optimal (lowest-cost) path from start to goal, and it is also optimally efficient: A* is guaranteed to expand (visit) fewer vertices in the graph than any other search algorithm, given that heuristic. Even with a non-admissible heuristic, A* will still find the goal, though it is not guaranteed to do so as efficiently as with an admissible heuristic. These properties make A* an important graph search algorithm for many applications, particularly in search-intensive fields such as artificial intelligence or robotics.

For further treatment of the theory behind A*, and proof of the above properties, see the discussions in [3, 7, 6, 4].

1.2 BGL Implementation

We have chosen to implement A* within the BGL (Boost Graph Library) [1, 8] framework. BGL is a C++ graph library based on the idea of *generic programming*, and makes heavy use of templates in order to provide genericity in its container and algorithm implementations.

In the spirit of this genericity, our implementation of A* is such that, in its default form, it does not provide the "shortest path search" functionality described above. In fact, it works in a fashion very similar to that of other search algorithms provided by BGL, such as breadth first search (BFS) and depth first search (DFS). Rather than require the specification of a goal vertex within the graph, our implementation takes only the graph itself and a heuristic function, and "searches" the graph, guided by the heuristic. In other words, it visits every vertex in the graph that belongs to the same connected component as

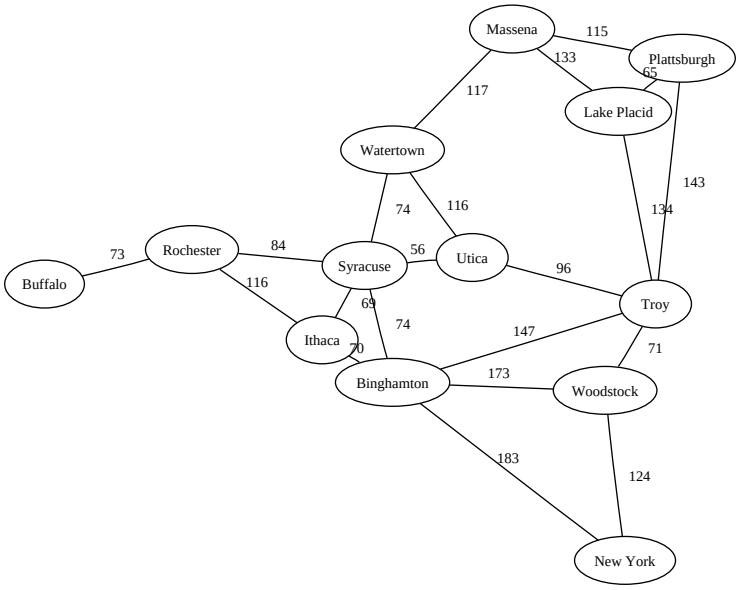

Figure 1: A "roadmap" of cities in New York State. We'd like to search to find the shortest path between any pair of cities in the map.

the start vertex, in an order that depends on the heuristic function. This approach is very similar to that used in BGL's BFS implementation. The functionality of our implementation can be easily extended to provide the shortest path search capability most commonly associated with A* (we give an example in Section 1.3), but can allow for other types of heuristic-guided searching as well.

Our A* implementation assumes a directed or undirected graph, possibly weighted. It makes use of a special visitor concept, AStarVisitor, and requires a heuristic function that is implemented as a model of the AStarHeuristic concept. A model of AStarHeuristic is a single-argument function object (functor) that returns a cost-to-goal estimate given some vertex in the graph being searched. For full details, see the reference manual (Section 2).

Note that, like with other BGL search algorithms, our A* implementation has no termination condition other than that it has visited every vertex in the same connected component as the start vertex. A custom visitor can cause the algorithm to terminate by throwing an exception.

1.3 Example Usage: Explicit Graphs

Often, graph search is discussed in the context of "path search". Suppose we have a "map", with information about places and paths between them. For example, consider the "roadmap" in Figure 1.3 of cities in New York State. The shortest path problem is to find the sequence of cities to visit (a "path" to follow) in order to most efficiently get from one location to another. In our

map, the "cost" to move between cities is specified in terms of time.

In this example, we create a BGL representation of the roadmap from Figure 1.3. We then pick two cities at random, and use A* to find the best path between them. An outline of the code is as follows:

"test-astar-cities.cpp" 5a ≡

```
⟨Include necessary files for path search 5b⟩
⟨Define auxiliary types 6⟩
⟨Specify a Euclidean distance heuristic 7⟩
⟨Specify a visitor that terminates upon finding the goal 8a⟩

int main(int argc, char **argv)
{
  ⟨Specify types and map data 8b⟩
  ⟨Create a BGL graph from the map data 9a⟩
  ⟨Pick random start and goal cities 9b⟩
  ⟨Output graph and start/goal 10a⟩
  ⟨Call the A* implementation 10b⟩
  ⟨If a path is found, print it to the screen 10c⟩
}
```

The A* search interface is specified in the include file astar_search.hpp. For this example, we use BGL's adjacency list graph representation. We also use Boost's random number generation functionaly (for picking random start and goal vertices).

⟨Include necessary files for path search 5b⟩ ≡

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/random.hpp>
#include <boost/random.hpp>
#include <boost/graph/graphviz.hpp>
#include <astar_search.hpp>
#include <sys/time.h>
#include <vector>
#include <list>
#include <iostream>
#include <fstream>
#include <math.h>    // for sqrt

using namespace boost;
using namespace std;
```

Used in part 5a.

In our map, we associate with each city a latitude and longitude, and use these in computing a Euclidean distance to the goal. This distance is used as our heuristic estimate of the true cost from a vertex to the goal. In order to associate latitude and longitude (y and x) values with each city, and specify the cost (in minutes) between two "adjacent" cities, we define auxilary types. Also defined here is a small class, for use with BGL's graphviz functionality, to write cities to a dot-format file. This class specifies locations (in points) for graph vertices, based on their latitude and longitude. One more class is provided to write edge weights to a dot-file.

⟨Define auxiliary types 6⟩ ≡

```
// auxiliary types
struct location
{
  float y, x; // lat, long
};
typedef float cost;

template <class Name, class LocMap>
class city_writer {
public:
  city_writer(Name n, LocMap l, float _minx, float _maxx,
              float _miny, float _maxy,
              unsigned int _ptx, unsigned int _pty)
    : name(n), loc(l), minx(_minx), maxx(_maxx), miny(_miny),
      maxy(_maxy), ptx(_ptx), pty(_pty) {}
  template <class Vertex>
  void operator()(ostream& out, const Vertex& v) const {
    float px = 1 - (loc[v].x - minx) / (maxx - minx);
    float py = (loc[v].y - miny) / (maxy - miny);
    out << "[label=\"" << name[v] << "\", pos=\""
        << static_cast<unsigned int>(ptx * px) << ","
        << static_cast<unsigned int>(pty * py)
        << "\", fontsize=\"11\"]";
  }
private:
  Name name;
  LocMap loc;
  float minx, maxx, miny, maxy;
  unsigned int ptx, pty;
};

template <class WeightMap>
class time_writer {
public:
  time_writer(WeightMap w) : wm(w) {}
  template <class Edge>
  void operator()(ostream &out, const Edge& e) const {
```

```
    out << "[label=\"" << wm[e] << "\", fontsize=\"11\"]";
  }
private:
  WeightMap wm;
};
```

Used in part 5a.

A heuristic function for our A* implementation is a model of the AStarHeuristic concept. It must define an operator() implementation that takes a single argument—a graph vertex—and that returns an estimated "cost" for that vertex. In this example, we estimate the cost from a vertex to the goal using the Euclidean distance between the vertex and the goal, calculated using the latitude and longitude of the two vertices. (Since the true cost between vertices is specified in terms of time, our heuristic is not strictly admissible since one might travel at more than sixty miles per hour! Here, we assume that this is not the case, making our heuristic is admissible.) Our heuristic is derived from the default astar_heuristic, which simply returns "zero" as its estimate. Note that the A* implementation does not require the specification of any goal vertex—the heuristic function object must itself incorporate this information.

⟨Specify a Euclidean distance heuristic 7⟩ ≡

```
// euclidean distance heuristic
template <class Graph, class CostType, class LocMap>
class distance_heuristic : public astar_heuristic<Graph, CostType>
{
public:
  typedef typename graph_traits<Graph>::vertex_descriptor Vertex;
  distance_heuristic(LocMap l, Vertex goal)
    : m_location(l), m_goal(goal) {}
  CostType operator()(Vertex u)
  {
    CostType dx = m_location[m_goal].x - m_location[u].x;
    CostType dy = m_location[m_goal].y - m_location[u].y;
    return ::sqrt(dx * dx + dy * dy);
  }
private:
  LocMap m_location;
  Vertex m_goal;
};
```

Used in part 5a.

Since we are looking for the shortest path between two vertices of the graph, it does not make sense for us to continue searching the graph after we've found the goal—once A* examines the goal vertex, it has found the shortest path if the heuristic being used is admissible! We define a special model of the AStarVisitor concept that throws an exception when A* examines the goal, terminating the algorithm.

⟨Specify a visitor that terminates upon finding the goal 8a⟩ ≡

```
struct found_goal {}; // exception for termination

// visitor that terminates when we find the goal
template <class Vertex>
class astar_goal_visitor : public boost::default_astar_visitor
{
public:
  astar_goal_visitor(Vertex goal) : m_goal(goal) {}
  template <class Graph>
  void examine_vertex(Vertex u, Graph& g) {
    if(u == m_goal)
      throw found_goal();
  }
private:
  Vertex m_goal;
};
```

Used in part 5a.

We now begin the main program of our example. First, we make several type definitions, including the graph type, that simplify our code later on. We then specify the actual data for our roadmap.

⟨Specify types and map data 8b⟩ ≡

```
// specify some types
typedef adjacency_list<listS, vecS, undirectedS, no_property,
  property<edge_weight_t, cost> > mygraph_t;
typedef property_map<mygraph_t, edge_weight_t>::type WeightMap;
typedef mygraph_t::vertex_descriptor vertex;
typedef mygraph_t::edge_descriptor edge_descriptor;
typedef mygraph_t::vertex_iterator vertex_iterator;
typedef std::pair<int, int> edge;

// specify data
enum nodes {
  Troy, LakePlacid, Plattsburgh, Massena, Watertown, Utica,
  Syracuse, Rochester, Buffalo, Ithaca, Binghamton, Woodstock,
  NewYork, N
};
const char *name[] = {
  "Troy", "Lake Placid", "Plattsburgh", "Massena",
  "Watertown", "Utica", "Syracuse", "Rochester", "Buffalo",
  "Ithaca", "Binghamton", "Woodstock", "New York"
};
location locations[] = { // lat/long
  {42.73, 73.68}, {44.28, 73.99}, {44.70, 73.46},
  {44.93, 74.89}, {43.97, 75.91}, {43.10, 75.23},
```

```
  {43.04, 76.14}, {43.17, 77.61}, {42.89, 78.86},
  {42.44, 76.50}, {42.10, 75.91}, {42.04, 74.11},
  {40.67, 73.94}
};
edge edge_array[] = {
  edge(Troy,Utica), edge(Troy,LakePlacid),
  edge(Troy,Plattsburgh), edge(LakePlacid,Plattsburgh),
  edge(Plattsburgh,Massena), edge(LakePlacid,Massena),
  edge(Massena,Watertown), edge(Watertown,Utica),
  edge(Watertown,Syracuse), edge(Utica,Syracuse),
  edge(Syracuse,Rochester), edge(Rochester,Buffalo),
  edge(Syracuse,Ithaca), edge(Ithaca,Binghamton),
  edge(Ithaca,Rochester), edge(Binghamton,Troy),
  edge(Binghamton,Woodstock), edge(Binghamton,NewYork),
  edge(Syracuse,Binghamton), edge(Woodstock,Troy),
  edge(Woodstock,NewYork)
};
unsigned int num_edges = sizeof(edge_array) / sizeof(edge);
cost weights[] = { // estimated travel time (mins)
  96, 134, 143, 65, 115, 133, 117, 116, 74, 56,
  84, 73, 69, 70, 116, 147, 173, 183, 74, 71, 124
};
```

Used in part 5a.

Given this data, we convert it to a BGL representation as follows:

⟨Create a BGL graph from the map data 9a⟩ ≡

```
// create graph
mygraph_t g(N);
WeightMap weightmap = get(edge_weight, g);
for(std::size_t j = 0; j < num_edges; ++j) {
  edge_descriptor e; bool inserted;
  tie(e, inserted) = add_edge(edge_array[j].first,
                              edge_array[j].second, g);
  weightmap[e] = weights[j];
}
```

Used in part 5a.

Next, we pick two cities at random from the graph. These will be our start and goal vertices.

⟨Pick random start and goal cities 9b⟩ ≡

```
// pick random start/goal
mt19937 gen(time(0));
vertex start = random_vertex(g, gen);
vertex goal = random_vertex(g, gen);
```

Used in part 5a.

Before calling the A* search algorithm, we output the graph to a dot-file, and write the start and goal cities to the standard output stream.

⟨Output graph and start/goal 10a⟩ ≡

```
cout << "Start vertex: " << name[start] << endl;
cout << "Goal vertex: " << name[goal] << endl;

ofstream dotfile;
dotfile.open("test-astar-cities.dot");
write_graphviz(dotfile, g,
               city_writer<const char **, location*>
                (name, locations, 73.46, 78.86, 40.67, 44.93,
                 480, 400),
               time_writer<WeightMap>(weightmap));
```

Used in part 5a.

Now, everything is ready for a graph search to be performed. We construct two vectors to use as property maps (one as a predecessor map, and the other to record actual distances between the start vertex and other vertices). Then, we call the named parameter interface to astar_search, constructing our heuristic function object and our visitor in the process.

⟨Call the A* implementation 10b⟩ ≡

```
vector<mygraph_t::vertex_descriptor> p(num_vertices(g));
vector<cost> d(num_vertices(g));
try {
  // call astar named parameter interface
  astar_search
    (g, start,
     distance_heuristic<mygraph_t, cost, location*>
      (locations, goal),
     predecessor_map(&p[0]).distance_map(&d[0]).
     visitor(astar_goal_visitor<vertex>(goal)));
```

Used in part 5a.

Finally, if we find a path to the goal, we construct a STL list representation of it by "walking" back up the predecessor tree, recorded in the predecessor map. We then print the path to the standard output stream.

⟨If a path is found, print it to the screen 10c⟩ ≡

```
  } catch(found_goal fg) { // found a path to the goal
    list<vertex> shortest_path;
    for(vertex v = goal;; v = p[v]) {
      shortest_path.push_front(v);
      if(p[v] == v)
        break;
    }
    cout << "Shortest path from " << name[start] << " to "
         << name[goal] << ": ";
    list<vertex>::iterator spi = shortest_path.begin();
    cout << name[start];
    for(++spi; spi != shortest_path.end(); ++spi)
      cout << " -> " << name[*spi];
    cout << endl << "Total travel time: " << d[goal] << endl;
    return 0;
  }

  cout << "Didn't find a path from " << name[start] << "to"
       << name[goal] << "!" << endl;
  return 0;
```

Used in part 5a.

Running this example several times, we obtain output as follows:

```
Start vertex: Watertown
Goal vertex: New York
Shortest path from Watertown to New York: Watertown ->
  Syracuse -> Binghamton -> New York
Total travel time: 331

Start vertex: Watertown
Goal vertex: Plattsburgh
Shortest path from Watertown to Plattsburgh: Watertown ->
  Massena -> Plattsburgh
Total travel time: 232

Start vertex: Rochester
Goal vertex: Rochester
Shortest path from Rochester to Rochester: Rochester
Total travel time: 0
```

This example has illustrated the use of our A* implementation for *explicit* (completely specified) graphs. We now turn our attention to the search of *implicit* graphs.

1.4 Example Usage: Implicit Graphs

Implicit graphs are graphs that are not completely specified (i.e., there is no representation of them stored completely in memory). Usually, vertices adjacent to some vertex are "generated" as necessary. Implicit graphs are particularly useful for representing very large spaces, for which is impractical to store the entire space in memory. For example, a chess-playing program cannot store every possible state of the chess board in memory—rather, it generates "children" of the current board state as a search tree, to some limited depth.

An example of searching implicit graphs with BGL is discussed in Chapter 9 of the *BGL User Guide and Reference Manual* [8]. Here, we give another example, using A*, in which we do things slightly differently.

In order to use implicit graphs with astar_search, we need to modify BGL's breadth-first search implementation slightly. This implementation (along with those of other algorithms in BGL) takes a `const Graph&` argument—which means that the graph cannot be changed by the search, or visitors to the search! Comments in the BFS implementation indicate that this is done so that graph adaptors can be passed. For now, we ignore the conflicts caused by removing the `const` requirement, in order to show how implicit graph searches might work. This is the only change we make to the BFS code. (The modified code is provided as nonconst_bfs.hpp.)

The 8-puzzle is a common example of a case in which implicit graph search is useful. In the 8-puzzle, a board with nine "cells" (eight of them filled with numbered tiles) is specified. The goal is to slide the tiles (using only the single free cell) into some specific configuration. The search space for the 8-puzzle is quite large.

Our code will actually be capable of solving the more general $(n^2 - 1)$-puzzle, though we only use it here with the 8-puzzle. Here is the outline for the 8-puzzle example code:

`"test-astar-8puzzle.cpp"` 12 ≡

```
⟨Include files for 8-puzzle example 13a⟩
⟨Define a class to represent a puzzle board state 13b⟩
⟨Define a function for generating the successors of a state 14⟩
⟨Define graph and related types 15⟩
⟨Define a visitor that generates puzzle states on the fly 16⟩
⟨Define a Manhattan Distance heuristic 17⟩

int main(int argc, char **argv)
{
  ⟨Set up start and goal puzzle states 18a⟩
  try {
    ⟨Call A* to search for the goal puzzle state 18b⟩
  } catch(found_goal fg) {
```

```
    ⟨Print the sequence of moves 18c⟩
  }
  accept(false);
  return 0;
}
```

We begin by including the necessary header files:

⟨Include files for 8-puzzle example 13a⟩ ≡

```
#include <nonconst_bfs.hpp> // so we can modify the graph
#include <algorithm>
#include <astar_search.hpp>
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include <list>
#include "test-astar-visitors.hpp"
#include "test-astar-accept.hpp"

using namespace boost;
using namespace std;
```

Used in parts 12, 52.

Note that we include nonconst_bfs.hpp, which modifies BGL's BFS so that we can use it with implicit graphs. To represent a board state (positions of each of the tiles), we define a simple class based on an STL vector. We also include an output operator for printing a board state to the screen.

⟨Define a class to represent a puzzle board state 13b⟩ ≡

```
class pstate_t : public vector<int>
{
public:
  int m_r, m_c;

  pstate_t() {}
  pstate_t(int rows, int cols)
    : vector<int>(m_r * m_c), m_r(rows), m_c(cols) {}
  template <class I>
  pstate_t(int rows, int cols, I beg, I end)
    : vector<int>(beg, end), m_r(rows), m_c(cols) {}

  inline int get(int r, int c) const {
    return operator[](cell(r, c));
  }
  inline void move(int i, int j) {
    int tmp = operator[](i);
    operator[](i) = operator[](j);
```

```
    operator[](j) = tmp;
  }
  // find offset of coordinates
  inline int cell(int r, int c) const {
    return r * m_c + c;
  }
  // find coordinates of an offset
  inline void coords(int i, int &r, int &c) const {
    r = i / m_c;
    c = i % m_c;
  }
};

ostream & operator<<(ostream &out, const pstate_t &p)
{
  for(int i = 0; i < p.m_r; ++i) {
    for(int j = 0; j < p.m_c; ++j) {
      if(p.get(i, j) > 0)
        cout << p.get(i, j) << " ";
      else
        cout << "  ";
    }
    cout << endl;
  }
  return out;
}
```

Used in parts 12, 52.

Our board state uses the cell containing zero as the "space". Based on the location of this space, we can slide the tiles immediately to the left, right, above and below into the space. Each of these "moves" constitutes a "child state"—i.e. a vertex in our search tree whose parent is the current state. The following function generates all possible children of a state:

⟨Define a function for generating the successors of a state 14⟩ ≡

```
void gen_children(const pstate_t &p, list<pstate_t> &children)
{
  pstate_t::const_iterator i = find(p.begin(), p.end(), 0);
  int sr, sc, soff = i - p.begin();
  p.coords(soff, sr, sc);
  if(sc > 0) { // move tile to left of space
    children.push_back(p);
    children.back().move(soff, p.cell(sr, sc - 1));
  }
  if(sc < p.m_c - 1) { // move tile to right of space
    children.push_back(p);
    children.back().move(soff, p.cell(sr, sc + 1));
  }
```

```
  if(sr > 0) { // move tile above space
    children.push_back(p);
    children.back().move(soff, p.cell(sr - 1, sc));
  }
  if(sr < p.m_r - 1) { // move tile below space
    children.push_back(p);
    children.back().move(soff, p.cell(sr + 1, sc));
  }
}
```

Used in parts 12, 52.

We now have defined everything necessary for working with the 8-puzzle, and we turn to the graph search itself. Since our graph is implicit (we will be modifying it as the search transpires), we *must* declare all property maps associated with the graph as *internal* maps, so that they grow properly with the graph. This includes the maps used internally by astar_search. In addition to these, we associate a puzzle state with each vertex in our graph.

⟨Define graph and related types 15⟩ ≡

```
struct vertex_puz_state_t {
  typedef vertex_property_tag kind;
};

typedef property<vertex_color_t, default_color_type,
  property<vertex_rank_t, unsigned int,
  property<vertex_distance_t, unsigned int,
  property<vertex_predecessor_t, unsigned int,
  property<vertex_puz_state_t, pstate_t> > > > > vert_prop;
typedef property<edge_weight_t, unsigned int> edge_prop;
typedef adjacency_list<listS, vecS, undirectedS, vert_prop,
  edge_prop> mygraph_t;
typedef mygraph_t::vertex_descriptor vertex_t;
typedef mygraph_t::vertex_iterator vertex_iterator_t;
typedef property_map<mygraph_t, vertex_puz_state_t>::type StateMap;
typedef property_map<mygraph_t, edge_weight_t>::type WeightMap;
typedef property_map<mygraph_t, vertex_predecessor_t>::type PredMap;
```

Used in parts 12, 52.

We now define a special visitor that adds successor vertices to the graph when a vertex is examined. This visitor also checks to be sure any state it adds isn't already in the graph, in order to save space. (We do this inefficiently here in order to keep the example simple.) Our visitor also checks to see if the goal state has been reached, and if so terminates the search. We derive the visitor from a generic VisitorType so that we can later use it in performance tests with BFS.

⟨Define a visitor that generates puzzle states on the fly 16⟩ ≡

```
struct found_goal {};

template <class VisitorType>
class puz_visitor
  : public examine_recorder<VisitorType, list<vertex_t> >
{
public:
  puz_visitor(pstate_t &goal, list<vertex_t> &seq)
    : examine_recorder<VisitorType, list<vertex_t> >(seq),
      m_goal(goal), m_seq(seq) {}

  template <class Vertex, class Graph>
  void examine_vertex(Vertex u, Graph& g) {
    examine_recorder<VisitorType, list<vertex_t> >
      ::examine_vertex(u, g);

    StateMap smap = get(vertex_puz_state_t(), g);

    // check for goal
    if(smap[u] == m_goal)
      throw found_goal();

    // add successors of this state
    list<pstate_t> children;
    gen_children(smap[u], children);
    list<pstate_t>::iterator i = children.begin();
    for(; i != children.end(); ++i) {
      // make sure this state is new (very inefficient this way)
      vertex_iterator_t vi, vend;
      for(tie(vi, vend) = vertices(g); vi != vend; ++vi)
        if(smap[*vi] == *i)
          break;
      if(vi != vend) // not new
        add_edge(u, *vi, edge_prop(1), g);
      else { // new
        vertex_t v = add_vertex(vert_prop(white_color), g);
        smap[v] = *i;
        add_edge(u, v, edge_prop(1), g);
      }
    }
  }
private:
  pstate_t &m_goal;
  list<vertex_t> &m_seq;
};
```

Used in parts 12, 52.

We now have nearly everything necessary to perform a search that finds a sequence of moves from the start state, that will lead to the goal state. In order to use A* for this search, we must specify a heuristic. Were we to use the default zero-heuristic, our search would be equivalent to a breadth-first search (since the costs of each edge in the graph, defined in terms of number of moves, are identically 1). One of the most commonly used heuristics for the 8-puzzle is the "Manhattan Distance" between a state and the goal. Manhattan distance is defined as

$$M(s,g) = \sum_{t \in \text{tiles}} |\text{row}(g_t) - \text{row}(s_t)| + |\text{col}(g_t) - \text{col}(s_t)|$$

In other words, for every tile in the state, we calculate how many rows and how many columns it is from its location in the goal state. The sume of the offsets for all of the tiles is the Manhattan distance.

⟨Define a Manhattan Distance heuristic 17⟩ ≡

```
// manhattan distance heuristic
class manhattan_dist
  : public astar_heuristic<mygraph_t, unsigned int>
{
public:
  manhattan_dist(pstate_t &goal, StateMap &smap)
    : m_goal(goal), m_smap(smap) {}
  unsigned int operator()(vertex_t u) {
    unsigned int md = 0;
    pstate_t::const_iterator i, j;
    int ir, ic, jr, jc;
    for(i = m_smap[u].begin(); i != m_smap[u].end(); ++i) {
      j = find(m_goal.begin(), m_goal.end(), *i);
      m_smap[u].coords(i - m_smap[u].begin(), ir, ic);
      m_goal.coords(j - m_goal.begin(), jr, jc);
      md += myabs(jr - ir) + myabs(jc - ic);
    }
    return md;
  }
private:
  pstate_t &m_goal;
  StateMap m_smap;
  inline unsigned int myabs(int i) {
    return static_cast<unsigned int>(i < 0 ? -i : i);
  }
};
```

Used in part 12.

All that remains is to specify the specific 8-puzzle problem we'd like to solve, and perform our search. We begin by adding a single start vertex to our graph,

and specifying the goal board state. (We provide several board states of varying difficulty.)

⟨Set up start and goal puzzle states 18a⟩ ≡

```
mygraph_t g;
list<vertex_t> examine_seq;
StateMap smap = get(vertex_puz_state_t(), g);
vertex_t start = add_vertex(vert_prop(white_color), g);
int sstart[] = {1, 3, 4, 8, 0, 2, 7, 6, 5}; // 4 steps
//  int sstart[] = {2, 8, 3, 1, 6, 4, 7, 0, 5}; // 5 steps
//  int sstart[] = {2, 1, 6, 4, 0, 8, 7, 5, 3}; // 18 steps
int sgoal[] = {1, 2, 3, 8, 0, 4, 7, 6, 5};
smap[start] = pstate_t(3, 3, &sstart[0], &sstart[9]);
pstate_t psgoal(3, 3, &sgoal[0], &sgoal[9]);
cout << "Start state:" << endl << smap[start] << endl;
cout << "Goal state:" << endl << psgoal << endl;
```

Used in parts 12, 52.

In order to perform the actual search, we instantiate our visitor and call astar_search, passing all necessary named parameters.

⟨Call A* to search for the goal puzzle state 18b⟩ ≡

```
puz_visitor<default_astar_visitor> vis(psgoal, examine_seq);
astar_search(g, start, manhattan_dist(psgoal, smap),
             visitor(vis).color_map(get(vertex_color, g)).
             rank_map(get(vertex_rank, g)).
             distance_map(get(vertex_distance, g)).
             predecessor_map(get(vertex_predecessor, g)));
```

Used in part 12.

If the search successfully finds a path to the goal (which it always should), we print out this path, along with the number of vertices examined by our search:

⟨Print the sequence of moves 18c⟩ ≡

```
PredMap p = get(vertex_predecessor, g);
list<vertex_t> shortest_path;
for(vertex_t v = examine_seq.back();; v = p[v]) {
  shortest_path.push_front(v);
  if(p[v] == v)
    break;
}
cout << "Sequence of moves:" << endl;
list<vertex_t>::iterator spi = shortest_path.begin();
for(; spi != shortest_path.end(); ++spi)
```

```
  cout << smap[*spi] << endl;
cout << "Number of moves: "
     << shortest_path.size() - 1 << endl;
cout << "Number of vertices examined: "
     << examine_seq.size() << endl;
return 0;
```

Used in part 12.

For a comparison of the performance of BFS and A* in solving the 8-puzzle, see Section 5.2.3.

2 Reference Manual

2.1 astar_search

⟨astar_search interfaces 19⟩ ≡

```
// Named parameter interface
template <typename VertexListGraph,
          typename AStarHeuristic,
          typename P, typename T, typename R>
void
astar_search
  (VertexListGraph &g,
   typename graph_traits<VertexListGraph>::vertex_descriptor s,
   AStarHeuristic h, const bgl_named_params<P, T, R>& params);

// Non-named parameter interface
template <typename VertexListGraph, typename AStarHeuristic,
          typename AStarVisitor, typename PredecessorMap,
          typename CostMap, typename DistanceMap,
          typename WeightMap, typename VertexIndexMap,
          typename ColorMap,
          typename CompareFunction, typename CombineFunction,
          typename CostInf, typename CostZero>
inline void
astar_search
  (VertexListGraph &g,
   typename graph_traits<VertexListGraph>::vertex_descriptor s,
   AStarHeuristic h, AStarVisitor vis,
   PredecessorMap predecessor, CostMap cost,
   DistanceMap distance, WeightMap weight,
   VertexIndexMap index_map, ColorMap color,
   CompareFunction compare, CombineFunction combine,
   CostInf inf, CostZero zero);
```

Not used.

This algorithm implements a heuristic search on a weighted, directed or undirected graph for the case where all edge weights are non-negative. For further information on the algorithm, see the User's Guide (Section 1).

The A* algorithm is very similar to Dijkstra's Shortest Paths algorithm. This implementation finds all the shortest paths from the start vertex to every other vertex by creating a search tree, examining vertices according to their remaining cost to some goal, as estimated by a heuristic function. Most commonly, A* is used to find some specific goal vertex or vertices in a graph, after which the search is terminated. An example of such usage is provided in the User's Guide.

For further details on the design and implementation of the BGL version of A*, see Design Issues (Section 3).

2.1.1 Where Defined

astar_search.hpp

2.1.2 Parameters

- IN: `VertexListGraph& g`

  The graph object on which the algorithm will be applied. The type VertexListGraph must be a model of the Vertex List Graph concept.

- IN: `vertex_descriptor s`

  The start vertex for the search. All distances will be calculated from this vertex, and the shortest paths tree (recorded in the predecessor map) will be rooted at this vertex.

- IN: `AStarHeuristic h`

  The heuristic function that guides the search. The type AStarHeuristic must be a model of the AStarHeuristic concept.

2.1.3 Named Parameters

- IN: `weight_map(WeightMap w_map)`

  The weight or "length" of each edge in the graph. The weights must all be non-negative; the algorithm will throw a negative_edge exception if one of the edges is negative. The type WeightMap must be a model of Readable Property Map. The edge descriptor type of the graph needs to be usable as the key type for the weight map. The value type for this map must be the same as the value type of the distance map.

  Default: `get(edge_weight, g)`

- IN: `vertex_index_map(VertexIndexMap i_map)`

This maps each vertex to an integer in the range [0, num_vertices(g)). This is necessary for efficient updates of the heap data structure when an edge is relaxed. The type VertexIndexMap must be a model of Readable Property Map. The value type of the map must be an integer type. The vertex descriptor type of the graph needs to be usable as the key type of the map.

Default: `get(vertex_index, g)`

- OUT: `predecessor_map(PredecessorMap p_map)`

  The predecessor map records the edges in the minimum spanning tree. Upon completion of the algorithm, the edges (p[u],u) for all u in V are in the minimum spanning tree. If p[u] = u then u is either the start vertex or a vertex that is not reachable from the start. The PredecessorMap type must be a Read/Write Property Map with key and vertex types the same as the vertex descriptor type of the graph.

  Default: `dummy_property_map`

- UTIL/OUT: `distance_map(DistanceMap d_map)`

  The shortest path weight from the start vertex s to each vertex in the graph g is recorded in this property map. The shortest path weight is the sum of the edge weights along the shortest path. The type DistanceMap must be a model of Read/Write Property Map. The vertex descriptor type of the graph needs to be usable as the key type of the distance map. The value type of the distance map is the element type of a Monoid formed with the combine function object and the zero object for the identity element. Also the distance value type must have a StrictWeakOrdering provided by the compare function object.

  Default: `iterator_property_map` created from a std::vector with the same value type as the WeightMap, and of size num_vertices(g), and using the i_map for the index map.

- UTIL/OUT: `rank_map(CostMap c_map)`

  The f-value for each vertex. The f-value is defined as the sum of the cost to get to a vertex from the start vertex, and the estimated cost (as returned by the heuristic function h) from the vertex to a goal. The type CostMap must be a model of Read/Write Property Map. The vertex descriptor type of the graph needs to be usable as the key type of the distance map. The value type of the distance map is the element type of a Monoid formed with the combine function object and the zero object for the identity element. Also the distance value type must have a StrictWeakOrdering provided by the compare function object. The value type for this map must be the same as the value type for the distance map.

  Default: `iterator_property_map` created from a std::vector with the same value type as the WeightMap, and of size num_vertices(g), and using the i_map for the index map.

- UTIL/OUT: `color_map(ColorMap c_map)`

  This is used during the execution of the algorithm to mark the vertices, indicating whether they are on the OPEN or CLOSED lists. The vertices start out white and become gray when they are inserted into the OPEN list. They then turn black when they are examined and placed on the CLOSED list. At the end of the algorithm, vertices reachable from the source vertex will have been colored black. All other vertices will still be white. The type ColorMap must be a model of Read/Write Property Map. A vertex descriptor must be usable as the key type of the map, and the value type of the map must be a model of Color Value.

  Default: `iterator_property_map` created from a std::vector of value type default_color_type, with size num_vertices(g), and using the i_map for the index map.

- IN: `distance_compare(CompareFunction cmp)`

  This function is use to compare distances to determine which vertex is closer to the start vertex, and to compare f-values to determine which vertex on the OPEN list to examine next. The CompareFunction type must be a model of Binary Predicate and have argument types that match the value type of the DistanceMap property map.

  Default: `std::less<D>` with
  `D = typename property_traits<DistanceMap>::value_type`.

- IN: `distance_combine(CombineFunction cmb)`

  This function is used to combine distances to compute the distance of a path, and to combine distance and heuristic values to compute the f-value of a vertex. The CombineFunction type must be a model of Binary Function. Both argument types of the binary function must match the value type of the DistanceMap property map (which is the same as that of the WeightMap and CostMap property maps). The result type must be the same type as the distance value type.

  Default: `std::plus<D>` with
  `D = typename property_traits<DistanceMap>::value_type`.

- IN: `distance_inf(D inf)`

  The inf object must be the greatest value of any D object. That is, `compare(d, inf) == true` for any `d != inf`. The type D is the value type of the DistanceMap.

  Default: `std::numeric_limits<D>::max()`

- IN: `distance_zero(D zero)`

  The zero value must be the identity element for the Monoid formed by the distance values and the combine function object. The type D is the value type of the DistanceMap.

Default: `D()` with
`D = typename property_traits<DistanceMap>::value_type`.

- OUT: `visitor(AStarVisitor v)`

  Use this to specify actions that you would like to happen during certain event points within the algorithm. The type AStarVisitor must be a model of the AStarVisitor concept. The visitor object is passed by value [1].

  Default: `astar_visitor<null_visitor>`

2.1.4 Complexity

The time complexity is $O((E + V) \log V)$.

2.2 AStarHeuristic Concept

This concept defines the interface for the heuristic function, which is responsible for estimating the remaining cost from some vertex to a goal. The user can create a class that matches this interface, and then pass objects of the class into astar_search() to guide the order of vertex examination of the search. The heuristic instance must incorporate any necessary information about goal vertices in the graph.

2.2.1 Refinement of

Unary Function (must take a single argument—a graph vertex—and return a cost value) and Copy Constructible (copying a heuristic should be a lightweight operation).

2.2.2 Notation

H	A type that is a model of AStarHeuristic.
h	An object of type H.
G	A type that is a model of Graph.
g	An object of type G.
e	An object of type boost::graph_traits<G>::edge_descriptor.
s,u,v	An object of type boost::graph_traits<G>::vertex_descriptor.
CostType	A type that can be used with the compare and combine functions passed to A*.
c	An object of type CostType.

2.2.3 Associated Types

None.

[1] Since the visitor parameter is passed by value, if your visitor contains state then any changes to the state during the algorithm will be made to a copy of the visitor object, not the visitor object passed in. Therefore you may want the visitor to hold this state by pointer or reference.

2.2.4 Valid Expressions

- **Name**: Call Heuristic
  Expression: `CostType c = h(u)`
  Return Type: `CostType`
  Description: Called for the target of every out edge of a vertex being examined.

2.2.5 Models

- astar_heuristic

2.3 AStarVisitor Concept

This concept defines the visitor interface for astar_search(). The user can create a class that matches this interface, and then pass objects of the class into astar_search() to augment the actions taken during the search.

2.3.1 Refinement of

Copy Constructible (copying a visitor should be a lightweight operation).

2.3.2 Notation

V	A type that is a model of AStarVisitor.
vis	An object of type V.
G	A type that is a model of Graph.
g	An object of type G.
e	An object of type boost::graph_traits<G>::edge_descriptor.
s,u,v	An object of type boost::graph_traits<G>::vertex_descriptor.
DistanceMap	A type that is a model of Read/Write Property Map.
d	An object of type DistanceMap.
WeightMap	A type that is a model of Readable Property Map.
w	An object of type WeightMap.

2.3.3 Associated Types

None.

2.3.4 Valid Expressions

All of these expressions have a `void` return type.

- **Name**: Initialize Vertex
  Expression: `vis.initialize_vertex(u, g)`
  Description: This is invoked on each vertex of the graph when it is first initialized (i.e., when its property maps are initialized).

- **Name**: Discover Vertex
  Expression: `vis.discover_vertex(u, g)`
  Description: This is invoked when a vertex is first discovered and is added to the OPEN list.

- **Name**: Examine Vertex
  Expression: `vis.examine_vertex(u, g)`
  Description: This is invoked when a vertex is popped from the queue (i.e., it has the lowest cost on the OPEN list).

- **Name**: Examine Edge
  Expression: `vis.examine_edge(e, g)`
  Description: This is invoked on every out edge of a vertex after it is examined.

- **Name**: Edge Relaxed
  Expression: `vis.edge_relaxed(e, g)`
  Description: Upon examination, if the following condition holds then the edge is relaxed (the distance of its target vertex is reduced) and this method is invoked:
  ```
  tie(u, s) = incident(e, g);
  D d_u = get(d, u), d_v = get(d, s);
  W w_e = get(w, e);
  assert(compare(combine(d_u, w_e), d_s));
  ```

- **Name**: Edge Not Relaxed
  Expression: `vis.edge_not_relaxed(e, g)`
  Description: Upon examination, if an edge is not relaxed (see above), then this method is invoked.

- **Name**: Black Target
  Expression: `vis.black_target(u, g)`
  Description: This is invoked when a vertex that is on the CLOSED list is "rediscovered" via a more efficient path, and is re-added to the OPEN list.

- **Name**: Finish Vertex
  Expression: `vis.finish_vertex(u, g)`
  Description: This is invoked on a vertex when it is added to the CLOSED list, which happens after all of its out edges have been examined.

2.3.5 Models

- astar_visitor

3 Design Issues

Following is an in-depth discussion of issues we have considered in the design of our implementation of A* within the BGL framework, and a detailed

explanation of the workings of our implementation.

3.1 Algorithm Overview

The A* algorithm is a graph search algorithm that is guaranteed to find the shortest path from one vertex to another within a graph, given a heuristic with certain properties. For a heuristic without these properties, A* is still a complete graph search algorithm (it is guaranteed to find the goal eventually, if any path to it exists). As such, the algorithm has many similarities to the other graph search algorithms implemented in BGL, as well as to BGL's shortest path algorithm implementations, like that for Dijkstra's algorithm.

It turns out, in fact, that we can base our implementation of A* on BGL's core BFS functionality; this reduces the complexity of our implementation and lends itself well to the promotion of genericity common throughout BGL. The key difference between A* and BFS is the use of the heuristic function in determining which node to visit next; by customizing the queue used by BFS, we attain this functionality.

A consequence of this approach is that our algorithm can be easily extended, primarily using enhanced "visitor" concepts. For example, we might create a more specific version of A* that returns the shortest path through a graph from one vertex to another (rather than just performing the search itself). Our core A* implementation is such that an extension of this nature is almost trivial to create. (For an example of such a visitor, see Section 1.3.)

3.2 Basic Formulation

From Section 1, we see that the most important arguments to the A* algorithm are:

- A weighted, directed or undirected graph G
- A heuristic function $h(v)$, where v is a vertex in G, that returns an estimate of the cost to get to our goal state from v
- A start (root) vertex within the graph, s
- A goal vertex g (or description of such a vertex), which may or may not be reachable from s (optional).

Our implementation of A* is based on an OPEN/CLOSED list formulation of the algorithm. In Algorithm 1 is high-level pseudocode for such a formulation, adapted from [3]. Vertices on the OPEN list have been "discovered" by the algorithm, but not "expanded" (we have not discovered their adjacent vertices). Vertices on the CLOSED list have been completely examined by our search (we have expanded them and added their children to the OPEN list). Vertices that are on neither list have not been encountered in any context so far in our search. A major advantage of this formulation of the A* algorithm over

Algorithm 1: **A*(G, s, g, h)**: OPEN/CLOSED list A* graph search formulation

```
place s on OPEN
while OPEN ≠ ∅ do
   v := min_{u∈OPEN} f(u) = c(s, v) + h(v)
   OPEN := OPEN −{v}
   CLOSED := CLOSED ∪{v}
   if v = g then
      return success
   end if
   for all u adjacent to v do
      compute f′(u)
      if u ∉ OPEN and u ∉ CLOSED then
         OPEN := OPEN ∪{u}
      else
         if f′(u) < f(u) then
            f(u) = f′(u)
            if u ∈ CLOSED then
               CLOSED := CLOSED −{u}
               OPEN := OPEN ∪{u}
            end if
         end if
      end if
   end for
end while
return failure
```

other approaches is that it avoids "cycles" in the state space; our search will not become trapped by loops in the graph.

3.3 Implementation Details

We choose to implement the OPEN/CLOSED lists not as actual "lists", but instead through the vertex coloring mechanisms provided by BGL. A vertex in OPEN is colored gray, and a vertex in CLOSED is colored black; a vertex that is yet to be discovered is colored white. Determining which list a vertex belongs to is thus an $O(1)$ operation. In fact, this coloring is taken care of mostly by the BFS implementation; the only special case we need to worry about occurs when a vertex is "rediscovered" via a more efficient path, in which case we need to remove the vertex from CLOSED and place it back on OPEN.

Our criteria for expanding a vertex v on the OPEN list is that it has the lowest $f(v) = g(v) + h(v)$ value of all vertices on OPEN. Thus, in addition to being able to quickly determine which list a vertex belongs to (if any), we must be able to quickly find the lowest-cost vertex on OPEN to expand. We cannot, however, simply maintain a pointer to the *current* lowest-cost vertex, since once we expand that vertex we'll need to know the next-lowest-cost vertex to expand. What we really need is a priority queue structure, containing all vertices in OPEN, sorted by $f(\cdot)$. Inserting items into this structure is $O(\log n)$, where, in the worst case, $n = V$ for our algorithm. We replace the non-priority queue used by default in BFS with this new priority queue. Specifically, we use the MutableQueue from the Boost Library, in much the same way it is used by the Dijkstra's implementation in BGL.

Clearly, in order to be able to store information about the f-value of a vertex, we need another property map (in addition to the color map). Initially, all f-values should be initialized to ∞, and new values should be assigned only when a new vertex is discovered. In fact, we keep two separate property maps, one associated with the known cost from the start vertex to the current vertex, and another associated with its f-value. This is necessary because, when discovering a new vertex, we must assign its known cost based on the known cost of its parent; were we to store only the f-value, we would lose this information.

So, internally, our implementation of the A* algorithm also requires:

- A color property map (for assigning colors to vertices)
- A cost/distance property map (for assigning known costs to get to each vertex from s)
- An f-value property map (for assigning f-values to vertices)
- A priority queue for choosing the vertex on OPEN with the lowest f-value

Now, we reimplement the A* algorithm with the formulation we have just described. See Algorithm 2 for a pseudocode version of this implementation.

Algorithm 2: **A*(G, s, h)**: BGL implementation

```
for each vertex u ∈ V do
    VISITOR: ⟨initialize vertex u⟩
    d[u] := f[u] := infinity
    color[u] := WHITE
    p[u] := u
end for
color[s] := GRAY
d[s] := 0
f[s] := h(s)
INSERT(Q, s)
VISITOR: ⟨discover vertex s⟩
while Q ≠ ∅ do
    u := EXTRACT-MIN(Q)
    VISITOR: ⟨examine vertex u⟩
    for each vertex v ∈ Adj[u] do
        VISITOR: ⟨examine edge (u, v)⟩
        if w(u, v) + d[u] < d[v] then
            VISITOR: ⟨edge (u, v) relaxed⟩
            d[v] := w(u, v) + d[u]
            f[v] := d[v] + h(v)
            p[v] := u
            if color[v] = WHITE then
                color[v] := GRAY
                INSERT(Q, v)
                VISITOR: ⟨discover vertex v⟩
            else if color[v] = BLACK then
                color[v] := GRAY
                INSERT(Q, v)
                VISITOR: ⟨reopen vertex v⟩
            end if
        else
            VISITOR: ⟨edge (u, v) not relaxed⟩
        end if
    end for
    color[u] := BLACK
    VISITOR: ⟨finish vertex u⟩
end while
```

In the pseudocode, w is the edge weight, d is the distance of a vertex from s, and Q is our priority queue, sorted by f, the estimated cost to the goal of the path through a vertex. p is a predecessor map.

Note that this function *does not require a goal vertex parameter*. The search is guided completely by the heuristic, which can itself incorporate information about a goal state if necessary. As such, there is no built-in termination condition; by default, the search continues until every vertex in the same connected component as the start has been examined.

A visitor can terminate the algorithm execution in the typical BGL manner, by throwing an exception in one of its event handlers.

3.4 Complexity

For our formulation of the A* algorithm, the space complexity is $O(V)$, since we must associate colors, known costs and f-values with each vertex using property maps, and must store up to V vertex references in the priority queue. The time complexity of the algorithm is $O((E + V) \log V)$, due to the $O(\log V)$ queue insertion operation.

4 Source Code

We now present the actual source code for our implementation of A*, and for the basic models of the AStarHeuristic and AStarVisitor concepts. Much of this code is based on BGL's Dijkstra's Shortest Paths implementation, which is also an extension of the breadth first search functionality in BGL.

4.1 A* Algorithm Implementation

First, we define the interface for the algorithm. As with most other BGL algorithm implementations, we provide two interfaces to the A* implementation: a "named parameter" version and a "non-named parameter" version. The non-named parameter interface requires the specification of all necessary arguments for the search, whereas the named parameter interface provides defaults for these, but allows these defaults to be overridden if necessary.

4.1.1 Non-named parameter interface

The non-named parameter interface is defined as follows:

⟨Define non-named parameter interface to astar_search 30⟩ ≡

⟨Non-named parameter interface implementation (post-initialization) 31b⟩

```
// Non-named parameter interface
template <typename VertexListGraph, typename AStarHeuristic,
          typename AStarVisitor, typename PredecessorMap,
```

```
          typename CostMap, typename DistanceMap,
          typename WeightMap, typename VertexIndexMap,
          typename ColorMap,
          typename CompareFunction, typename CombineFunction,
          typename CostInf, typename CostZero>
inline void
astar_search
  (VertexListGraph &g,
   typename graph_traits<VertexListGraph>::vertex_descriptor s,
   AStarHeuristic h, AStarVisitor vis,
   PredecessorMap predecessor, CostMap cost,
   DistanceMap distance, WeightMap weight,
   VertexIndexMap index_map, ColorMap color,
   CompareFunction compare, CombineFunction combine,
   CostInf inf, CostZero zero)
{
  ⟨Non-named parameter interface implementation (initialization) 31a⟩
}
```

Used in part 54.

For a discussion of the parameters to the astar_search function, see the Reference Manual (Section 2). The implementation of the non-named parameter interface is divided into two parts. The first initializes the property maps (including the distance, f-value and predecessor maps, along with the vertex coloring), and then calls the second (post-initialization) part.

⟨Non-named parameter interface implementation (initialization) 31a⟩ ≡

```
typename graph_traits<VertexListGraph>::vertex_iterator ui, ui_end;
for (tie(ui, ui_end) = vertices(g); ui != ui_end; ++ui) {
  put(distance, *ui, inf);
  put(cost, *ui, inf);
  put(predecessor, *ui, *ui);
}
put(distance, s, zero);
put(cost, s, h(s));

astar_search_no_init
  (g, s, h, vis, predecessor, cost, distance, weight,
   color, index_map, compare, combine, inf, zero);
```

Used in part 30.

The second part of the non-named parameter implementation (called by the first part) instantiates a MutableQueue (an updatable priority queue), and then calls BGL's breadth_first_visit using this queue and a special BFS visitor that implements A* (see Section 4.1.3 below).

⟨Non-named parameter interface implementation (post-initialization) 31b⟩ ≡

```
template <typename VertexListGraph, typename AStarHeuristic,
          typename AStarVisitor, typename PredecessorMap,
          typename CostMap, typename DistanceMap,
          typename WeightMap, typename ColorMap,
          typename VertexIndexMap,
          typename CompareFunction, typename CombineFunction,
          typename CostInf, typename CostZero>
inline void
astar_search_no_init
  (VertexListGraph &g,
   typename graph_traits<VertexListGraph>::vertex_descriptor s,
   AStarHeuristic h, AStarVisitor vis,
   PredecessorMap predecessor, CostMap cost,
   DistanceMap distance, WeightMap weight,
   ColorMap color, VertexIndexMap index_map,
   CompareFunction compare, CombineFunction combine,
   CostInf inf, CostZero zero)
{
  typedef indirect_cmp<CostMap, CompareFunction> IndirectCmp;
  IndirectCmp icmp(cost, compare);

  typedef typename graph_traits<VertexListGraph>::vertex_descriptor
    Vertex;
  typedef mutable_queue<Vertex, std::vector<Vertex>,
      IndirectCmp, VertexIndexMap>
    MutableQueue;
  MutableQueue Q(num_vertices(g), icmp, index_map);

  detail::astar_bfs_visitor<AStarHeuristic, AStarVisitor,
      MutableQueue, PredecessorMap, CostMap, DistanceMap,
      WeightMap, ColorMap, CombineFunction, CompareFunction>
    bfs_vis(h, vis, Q, predecessor, cost, distance, weight,
            color, combine, compare, zero);

  breadth_first_visit(g, s, Q, bfs_vis, color);
}
```

Used in part 30.

4.1.2 Named parameter interface

The named parameter interface is defined like so:

⟨Define named parameter interface to astar_search 32⟩ ≡

⟨Detail code for named parameter interface to astar_search 55b⟩

```
// Named parameter interface
template <typename VertexListGraph,
```

```
          typename AStarHeuristic,
          typename P, typename T, typename R>
void
astar_search
  (VertexListGraph &g,
   typename graph_traits<VertexListGraph>::vertex_descriptor s,
   AStarHeuristic h, const bgl_named_params<P, T, R>& params)
{
  ⟨Named parameter interface implementation 33a⟩
}
```

Used in part 54.

Most of the named parameter interface's implementation involves setting up proper defaults for unnamed parameters. The details for this are provided in Appendix A.1. The basic implementation of the astar_search function's named parameter interface just calls a function that performs this detailed setup:

⟨Named parameter interface implementation 33a⟩ ≡

```
detail::astar_dispatch1
  (g, s, h,
   get_param(params, vertex_rank),
   get_param(params, vertex_distance),
   choose_const_pmap(get_param(params, edge_weight), g, edge_weight),
   choose_const_pmap(get_param(params, vertex_index), g, vertex_index),
   get_param(params, vertex_color),
   params);
```

Used in part 32.

4.1.3 BFS Visitor

Our A* implementation is an extension of BGL's breadth first search, realized as a special BFS visitor that modifies the way the BFS queue is updated. This visitor is largely the same as the default BFS visitor, except for the implementations of the tree_edge, gray_target and black_target events. This visitor is defined as follows:

⟨Define BFS visitor that implements A* 33b⟩ ≡

```
template <class AStarHeuristic, class UniformCostVisitor,
          class UpdatableQueue, class PredecessorMap,
          class CostMap, class DistanceMap, class WeightMap,
          class ColorMap, class BinaryFunction,
          class BinaryPredicate>
struct astar_bfs_visitor
{
  ⟨BFS visitor type definitions 57a⟩
```

```
    ⟨BFS visitor constructor 57b⟩
    ⟨Basic (mostly "passthru") events 57c⟩
    ⟨Core BFS visitor events that implement A* 34a⟩
    ⟨BFS visitor member variables 58a⟩
};
```

Used in part 54.

Here we present only the core events that modify BFS so that it searches according to the A* algorithm. The remainder of the BFS visitor code is specified in Appendix A.1.2.

⟨Core BFS visitor events that implement A* 34a⟩ ≡

```
⟨BFS tree_edge event 34b⟩
⟨BFS gray_target event 35a⟩
⟨BFS black_target event 35b⟩
```

Used in part 33b.

The tree_edge event corresponds to the "discovery" of a new vertex. This vertex is added to the OPEN list (it is colored gray) in the BFS implementation. Here, we calculate the f-value (estimated cost) for the vertex.

⟨BFS tree_edge event 34b⟩ ≡

```
template <class Edge, class Graph>
void tree_edge(Edge e, Graph& g) {
  m_decreased = relax(e, g, m_weight, m_predecessor, m_distance,
                      m_combine, m_compare);
  if(m_decreased) {
    m_vis.edge_relaxed(e, g);
    put(m_cost, target(e, g),
        m_combine(get(m_distance, target(e, g)),
                  m_h(target(e, g))));
  } else
    m_vis.edge_not_relaxed(e, g);
}
```

Used in part 34a.

The gray_target event corresponds to the situation in which we "rediscover" a vertex that is currently in the OPEN list, via a new "path". If the cost to get to this vertex along the new path is less than the old cost, we update the f-value of the vertex and update the position of the vertex in the OPEN list's priority queue.

⟨BFS gray_target event 35a⟩ ≡

```
template <class Edge, class Graph>
void gray_target(Edge e, Graph& g) {
  m_decreased = relax(e, g, m_weight, m_predecessor, m_distance,
                      m_combine, m_compare);
  if(m_decreased) {
    put(m_cost, target(e, g),
        m_combine(get(m_distance, target(e, g)),
                  m_h(target(e, g))));
    m_Q.update(target(e, g));
    m_vis.edge_relaxed(e, g);
  } else
    m_vis.edge_not_relaxed(e, g);
}
```

Used in part 34a.

The black_target event occurs when we find a new path to a vertex on the CLOSED list. If our new path decreases the cost to this vertex, we need to remove it from the CLOSED list and place it back on the OPEN list.

⟨BFS black_target event 35b⟩ ≡

```
template <class Edge, class Graph>
void black_target(Edge e, Graph& g) {
  m_decreased = relax(e, g, m_weight, m_predecessor, m_distance,
                      m_combine, m_compare);
  if(m_decreased) {
    m_vis.edge_relaxed(e, g);
    put(m_cost, target(e, g),
        m_combine(get(m_distance, target(e, g)),
                  m_h(target(e, g))));
    m_Q.push(target(e, g));
    put(m_color, target(e, g), Color::gray());
    m_vis.black_target(e, g);
  } else
    m_vis.edge_not_relaxed(e, g);
}
```

Used in part 34a.

4.2 Basic AStarHeuristic

The AStarHeuristic concept, described in the Reference Manual (Section 2), defines a unary function object that returns an estimate of the cost from some vertex to a goal. Since the A* implementation itself knows nothing of goal vertices, the heuristic is responsible for incorporating such information (if necessary). We provide a concept-checking class for AStarHeuristic that verifies whether a class is a proper model of the concept:

⟨Concept check for AStarHeuristic concept 36a⟩ ≡

```
template <class Heuristic, class Graph>
struct AStarHeuristicConcept {
  void constraints()
  {
    function_requires< CopyConstructibleConcept<Heuristic> >();
    h(u);
  }
  Heuristic h;
  typename graph_traits<Graph>::vertex_descriptor u;
};
```

Used in part 54.

We also specify a default model of the concept, called astar_heuristic, that returns "zero" as the estimated cost for a vertex:

⟨Base astar_heuristic class 36b⟩ ≡

```
template <class Graph, class CostType>
class astar_heuristic : public std::unary_function<
  typename graph_traits<Graph>::vertex_descriptor, CostType>
{
public:
  typedef typename graph_traits<Graph>::vertex_descriptor Vertex;
  astar_heuristic() {}
  CostType operator()(Vertex u) { return static_cast<CostType>(0); }
};
```

Used in part 54.

4.3 Basic AStarVisitor

The AStarVisitor concept, also described in the Reference Manual, is very similar to the BFS visitor concept. The situation in which each event occurs is described in the Reference Manual, as well as in Algorithm 2. In addition to the events of a BFS visitor, the AStarVisitor concept defines edge_relaxed and edge_not_relaxed events (since in some circumstances A* does not add an edge in the graph to its search tree).

⟨Concept check for AStarVisitor concept 36c⟩ ≡

```
template <class Visitor, class Graph>
struct AStarVisitorConcept {
  void constraints()
  {
    function_requires< CopyConstructibleConcept<Visitor> >();
    vis.initialize_vertex(u, g);
```

```
    vis.discover_vertex(u, g);
    vis.examine_vertex(u, g);
    vis.examine_edge(e, g);
    vis.edge_relaxed(e, g);
    vis.edge_not_relaxed(e, g);
    vis.black_target(e, g);
    vis.finish_vertex(u, g);
  }
  Visitor vis;
  Graph g;
  typename graph_traits<Graph>::vertex_descriptor u;
  typename graph_traits<Graph>::edge_descriptor e;
};
```

Used in part 54.

We provide a base implementation of the AStarVisitor concept, as follows:

⟨Base astar_visitor class 37⟩ ≡

```
template <class Visitors = null_visitor>
class astar_visitor : public bfs_visitor<Visitors> {
public:
  astar_visitor() {}
  astar_visitor(Visitors vis)
    : bfs_visitor<Visitors>(vis) {}

  template <class Edge, class Graph>
  void edge_relaxed(Edge e, Graph& g) {
    invoke_visitors(this->m_vis, e, g, on_edge_relaxed());
  }
  template <class Edge, class Graph>
  void edge_not_relaxed(Edge e, Graph& g) {
    invoke_visitors(this->m_vis, e, g, on_edge_not_relaxed());
  }
private:
  template <class Edge, class Graph>
  void tree_edge(Edge e, Graph& g) {}
  template <class Edge, class Graph>
  void non_tree_edge(Edge e, Graph& g) {}
};
template <class Visitors>
astar_visitor<Visitors>
make_astar_visitor(Visitors vis) {
  return astar_visitor<Visitors>(vis);
}
typedef astar_visitor<> default_astar_visitor;
```

Used in part 54.

5 Test Plan and Results

In testing our A* algorithm for correctness and performance, we can make use of the implementations of similar algorithms already provided by BGL, to some extent. However, much of the correctness of our implementation is not easily verified in this manner.

All of the following tests were performed on a Pentium-III class laptop with 192MB of memory, running Linux 2.4.19. All programs were compiled with the gcc 3.2 compiler.

5.1 Correctness Testing

In our correctness testing, we must verify that the algorithm implementation itself is correct. We must also verify that the implementation meets the specified requirements—for example, it must work with both vecS and listS storage of vertices.

5.1.1 Comparison with an alternate implementation

The first of our tests compares the output of astar_search with that of another implementation of A*. The implementation we compare against is a modification of that in [2], which is known to be correct. For the code of this alternate A*, see Appendix A.3.

In our correctness test, we generate random weighted graphs, and then search these graph using both astar_search and the alternate implementation. We consider astar_search to be correct if the computed shortest-path distances from the start vertex to every other vertex are the same as those computed by the alternate implementation, and if both implementations examine precisely the same number of vertices over the course of their search.

"test-astar-alternate.cpp" 38 ≡

```
#include "alternate_astar.hpp"
⟨Include header files for random graph tests 39a⟩

⟨Define an "adaptor" class for using alternate A* 39b⟩

int main(int argc, char **argv)
{
  ⟨Read vertex and edge count arguments from the command line 40a⟩
  ⟨Declare a graph type for random graph tests 40b⟩
  ⟨Generate a random graph 41a⟩
  ⟨Pick a random start vertex 41b⟩
  ⟨Randomly assign weights to edges 41c⟩
  ⟨Output random graph information 41d⟩
```

```
  ⟨Call our version of A* 42a⟩
  ⟨Call the alternate version of A* 42b⟩
  ⟨Verify that the results are the same 42c⟩
  return 0;
}
```

The same files are included for most of our tests (both correctness and performance) involving random graphs.

⟨Include header files for random graph tests 39a⟩ ≡

```
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/random.hpp>
#include <boost/random.hpp>
#include <boost/random/uniform_smallint.hpp>
#include <astar_search.hpp>
#include <sys/time.h>
#include <vector>
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "test-astar-visitors.hpp"
#include "test-astar-accept.hpp"

using namespace boost;
using namespace std;
```

Used in parts 38, 43a, 44b, 45e, 47a, 51a.

In order to use the alternate A* implementation with BGL graphs, we need to define a simple "adaptor" class that provides the interface expected by the implementation. Essentially, this class is just "wrapped" around a BGL graph and an AStarHeuristic. It also counts the number of vertex examinations, for later use in our acceptance testing.

⟨Define an "adaptor" class for using alternate A* 39b⟩ ≡

```
template <typename Graph, typename AStarHeuristic,
          typename WeightMap>
class adaptor_t
{
public:
  typedef typename graph_traits<Graph>::vertex_descriptor
    V;
  typedef typename graph_traits<Graph>::out_edge_iterator
    e_iter;
  adaptor_t(Graph &g, AStarHeuristic h, WeightMap w)
    : m_count(0), m_g(g), m_h(h), m_weight(w) {}
```

```
  cost_t getNodeCost(V v)
  {
    return m_h(v);
  }
  void getNodeNeighbors(V v, list< pair<V, cost_t> > &n)
  {
    ++m_count;
    e_iter ei, eend;
    for(tie(ei, eend) = out_edges(v, m_g); ei != eend; ++ei)
      n.push_back(pair<V, cost_t>(target(*ei, m_g),
                                  m_weight[*ei]));
  }
  unsigned int examine_count() const { return m_count; }
private:
  unsigned int m_count;
  Graph m_g;
  AStarHeuristic m_h;
  const WeightMap m_weight;
};
```

Used in part 38.

At the beginning of our program, we read two optional arguments from the command line (if they are provided), for the number of vertices and number of edges, respectively. This makes it easy to run a variety of tests using shell scripts that run the code with different-sized graphs. This code is also used throughout our random graph testing.

⟨Read vertex and edge count arguments from the command line 40a⟩ ≡

```
unsigned int num_v = 40, num_e = 100;
if(argc > 1)
  num_v = atoi(argv[1]);
if(argc > 2)
  num_e = atoi(argv[2]);
if(!num_v) {
  cout << "Usage: " << argv[0]
       << " [num nodes (d. 40)] [num edges (d. 100)]"
       << endl;
  return 1;
}
```

Used in parts 38, 43a, 44b, 45e.

For most of our tests, it is convenient to use a vecS vertex list representation (for an example of using listS, see below). We also require an internal edge weight property (one of astar_search's requirements is that it be run on a weighted graph).

⟨Declare a graph type for random graph tests 40b⟩ ≡

```
typedef adjacency_list<listS, vecS, undirectedS, no_property,
  property<edge_weight_t, unsigned int> > mygraph_t;
typedef mygraph_t::vertex_descriptor vertex_t;
typedef mygraph_t::edge_iterator edge_iterator_t;
```

Used in parts 38, 44b, 45e, 51a.

We make use of BGL's random graph functionality to generate a random graph of the desired size:

⟨Generate a random graph 41a⟩ ≡

```
mt19937 gen(time(0));
mygraph_t g;
generate_random_graph(g, num_v, num_e, gen, false);
```

Used in parts 38, 43a, 44b, 45e, 51a.

We then pick a vertex at random to serve as the root of our search tree:

⟨Pick a random start vertex 41b⟩ ≡

```
vertex_t start = random_vertex(g, gen);
```

Used in parts 38, 43a, 44b, 45e, 51a.

We also randomly assign weights to the edges of our graph:

⟨Randomly assign weights to edges 41c⟩ ≡

```
uniform_smallint<mt19937, unsigned int> ugen(gen, 0, 9);
randomize_property<edge_weight_t>(g, ugen);
```

Used in parts 38, 43a, 45e, 51a.

For debugging purposes, it is often useful to view the randomly-generated graph. We write the graph in graphviz format to an output file.

⟨Output random graph information 41d⟩ ≡

```
cout << "Start vertex: " << start << endl;
ofstream dotfile;
dotfile.open("test-astar-alternate.dot");
write_graphviz(dotfile, g, default_writer(),
               make_label_writer(get(edge_weight, g)));
```

Used in part 38.

We first call astar_search and record its results, as follows:

⟨Call our version of A* 42a⟩ ≡

```
vector<unsigned int> ad(num_vertices(g));
list<vertex_t> aseq;
examine_recorder<default_astar_visitor,
  list<vertex_t> > avis(aseq);
astar_search(g, start,
             astar_heuristic<mygraph_t, unsigned int>(),
             distance_map(&ad[0]).
             visitor(avis));
```

Used in part 38.

We then construct an "adaptor" around our randomly-generated graph, and call the alternate implementation of A*, like so:

⟨Call the alternate version of A* 42b⟩ ≡

```
vector<unsigned int> bd(num_vertices(g));
adaptor_t<mygraph_t, astar_heuristic<mygraph_t, cost_t>,
  property_map<mygraph_t, edge_weight_t>::type>
  adaptor(g, astar_heuristic<mygraph_t, cost_t>(),
          get(edge_weight, g));
graphSearch(adaptor, start, bd);
```

Used in part 38.

As previously stated, astar_search is correct if it produces shortest-path distances equal to those of the alternate implementation, and if both implementations examine the same number of vertices. We check for this as follows:

⟨Verify that the results are the same 42c⟩ ≡

```
accept(aseq.size() == adaptor.examine_count());
vector<vertex_t>::iterator i = ad.begin(), j = bd.begin();
for(i = ad.begin(); i != ad.end(); ++i, ++j)
  accept(*i == *j || (*j == 0 &&
    *i == numeric_limits<unsigned int>::max()));
```

Used in part 38.

All of the above tests pass on graphs of varying complexity.

5.1.2 vecS and listS

Most of our examples use BGL's vecS vertex list representation, since the ability to treat vertices as integers makes some operations significantly simpler. Our A* implementation is incorrect though, if it does not also work with a listS representation. Our test for this is very simple: we just create a graph that uses a listS representation and run astar_search on it. The outline of the test is as follows:

"test-astar-list.cpp" 43a ≡

```
⟨Include header files for random graph tests 39a⟩

int main(int argc, char **argv)
{
  ⟨Read vertex and edge count arguments from the command line 40a⟩
  ⟨Declare a listS-based graph type 43b⟩
  ⟨Generate a random graph 41a⟩
  ⟨Pick a random start vertex 41b⟩
  ⟨Randomly assign weights to edges 41c⟩
  ⟨Initialize vertex index map 43c⟩
  ⟨Run astar_search on the listS-based graph 44a⟩
  return 0;
}
```

If this test compiles and runs, the A* implementation works properly with listS-based graphs. Note that most of the code for this test is the same as for our other random-graph tests. However, the graph type we use is different, defined as follows:

⟨Declare a listS-based graph type 43b⟩ ≡

```
typedef adjacency_list<listS, listS, undirectedS,
  property<vertex_index_t, int>,
  property<edge_weight_t, unsigned int> > mygraph_t;
typedef mygraph_t::vertex_descriptor vertex_t;
```

Used in part 43a.

This type uses a list to store vertices (rather than a vector). It also has a vertex index property map—so that we can map vertices in the list to integer indices for our other property maps. We initialize this vertex index map list so:

⟨Initialize vertex index map 43c⟩ ≡

```
property_map<mygraph_t, vertex_index_t>::type indexmap
  = get(vertex_index, g);
graph_traits<mygraph_t>::vertex_iterator vi, vend;
int c = 0;
for(tie(vi, vend) = vertices(g); vi != vend; ++vi, ++c)
  indexmap[*vi] = c;
```

Used in part 43a.

We then run astar_search on our graph as follows:

⟨Run astar_search on the listS-based graph 44a⟩ ≡

```
astar_search(g, start, astar_heuristic<mygraph_t, unsigned int>(),
             vertex_index_map(indexmap));
```

Used in part 43a.

This test compiles and runs. So long as a vertex index map is provided, our A* implementation works well with listS-based graphs.

5.1.3 BFS and Dijkstra tests

For uniformly weighted graphs, and using a zero-heuristic, we expect A* and breadth-first search to find equivalent-length paths from the start vertex to all vertices in a graph. A* and BFS may create a different search tree due to differences in the queues they use, and for the same reason they may visit the vertices of the graph in a different order. However, we can easily check that the distance maps produced by the two searches are the same. If they are not, our A* implementation is incorrect.

We once again generate random graphs. This time, we give the edges in the graph uniform weight. We run both astar_search and breadth_first_search on the graphs, and verify that the distance maps they output are the same. The outline of this test is as follows:

"test-astar-bfs.cpp" 44b ≡

```
⟨Include header files for random graph tests 39a⟩

int main(int argc, char **argv)
{
  ⟨Read vertex and edge count arguments from the command line 40a⟩
  ⟨Declare a graph type for random graph tests 40b⟩
  ⟨Generate a random graph 41a⟩
  ⟨Pick a random start vertex 41b⟩
  ⟨Assign uniform weights to edges 45a⟩
  ⟨Run astar_search and record distances 45b⟩
  ⟨Run breadth_first_search and record distances 45c⟩
  ⟨Verify that A* and BFS distance maps are the same 45d⟩
  return 0;
}
```

Once again, we reuse much of the code from our other random graph examples. We assign uniform weight to the edges in the graph with the following code:

⟨Assign uniform weights to edges 45a⟩ ≡

```
edge_iterator_t ei, eend;
for(tie(ei, eend) = edges(g); ei != eend; ++ei)
  put(get(edge_weight, g), *ei, 1);
```

Used in part 44b.

We then run astar_search and breadth_first_search on the graph, recording distances into separate distance maps:

⟨Run astar_search and record distances 45b⟩ ≡

```
vector<vertex_t> ad(num_vertices(g));
astar_search(g, start, astar_heuristic<mygraph_t, unsigned int>(),
             distance_map(&ad[0]));
```

Used in parts 44b, 45e.

⟨Run breadth_first_search and record distances 45c⟩ ≡

```
vector<vertex_t> bd(num_vertices(g));
breadth_first_search(g, start,
                     visitor(make_bfs_visitor
                             (record_distances
                              (&bd[0], on_tree_edge()))));
```

Used in part 44b.

In order to verify that the A* and BFS distance maps are the same, we just iterate through them and compare. One other consideration is that BFS assigns a distance of zero to any vertex that can't be reached from the start, whereas A* assigns *infinity* (in this case `std::numeric_limits<unsigned int>::max()`). So, we also check for this condition.

⟨Verify that A* and BFS distance maps are the same 45d⟩ ≡

```
vector<vertex_t>::iterator i = ad.begin(), j = bd.begin();
// if *j is zero, astar records this as infinity so *i != *j
for(; i != ad.end(); ++i, ++j)
  accept(*i == *j || (*j == 0 &&
    *i == numeric_limits<unsigned int>::max()));
```

Used in part 44b.

This test passes. We perform a similar test with Dijkstra's Shortest Paths algorithm and weighted graphs—in this case, the distance maps should also be the same. The outline of the Dijkstra test is as follows:

"test-astar-dijkstra.cpp" 45e ≡

```
#include <boost/graph/dijkstra_shortest_paths.hpp>
⟨Include header files for random graph tests 39a⟩

int main(int argc, char **argv)
{
  ⟨Read vertex and edge count arguments from the command line 40a⟩
  ⟨Declare a graph type for random graph tests 40b⟩
  ⟨Generate a random graph 41a⟩
  ⟨Pick a random start vertex 41b⟩
  ⟨Randomly assign weights to edges 41c⟩
  ⟨Run astar_search and record distances 45b⟩
  ⟨Run dijkstra_shortest_paths and record distances 46a⟩
  ⟨Verify that A* and Dijkstra distance maps are the same 46b⟩
  return 0;
}
```

The only differences between this and the previous test is that we randomly assign edge weights, and that we call the Dijkstra's Shortest Paths implementation. This call is just:

⟨Run dijkstra_shortest_paths and record distances 46a⟩ ≡

```
vector<vertex_t> bd(num_vertices(g));
dijkstra_shortest_paths(g, start, distance_map(&bd[0]));
```

Used in part 45e.

We verify that the distance maps are the same with:

⟨Verify that A* and Dijkstra distance maps are the same 46b⟩ ≡

```
vector<vertex_t>::iterator i = ad.begin(), j = bd.begin();
for(; i != ad.end(); ++i, ++j)
  accept(*i == *j);
```

Used in part 45e.

This test also passes.

5.1.4 Implicit graphs

One important use of A* and other graph search algorithms is for the searching of implicitly-defined graphs (see Section 1.4). Since A* relies heavily on property maps, we should check that modifying the graph at search-time does

not adversely affect our search. We do this by verifying that the sequence in which we visit vertices in an implicitly generated graph is the same as when that graph is explicitly defined.

As discussed in Section 1.4, BGL's BFS implementation does not allow online modification of graphs. We use a slightly modified version with which implicit graphs are possible. The outline of this test is as follows:

"test-astar-implicit.cpp" 47a ≡

```
#include <nonconst_bfs.hpp> // so we can modify the graph
⟨Include header files for random graph tests 39a⟩

unsigned int num_v = 40, cur_v = 0;
⟨Define implicit graph type 47b⟩
⟨Define visitor that generates a random tree on the fly 48a⟩

int main(int argc, char **argv)
{
  ⟨Read vertex count argument from the command line 48b⟩
  ⟨Declare a graph and add a single start vertex 49a⟩
  ⟨Run astar_search on the implicit graph 49b⟩
  ⟨Run astar_search again on the explicit graph 49c⟩
  ⟨Verify that examination sequences are the same 49d⟩
  return 0;
}
```

In order to use implicit graphs with A*, the following properties *must be available as internal property maps*:

- vertex_color_t
- vertex_rank_t
- vertex_distance_t
- edge_weight_t

This is because as the graph grows, external property maps do not. These properties are central to the A* algorithm's correctness, and their maps must grow with the graph. So, we define our implicit graph type as:

⟨Define implicit graph type 47b⟩ ≡

```
typedef property<vertex_color_t, default_color_type,
  property<vertex_rank_t, unsigned int,
    property<vertex_distance_t, unsigned int> > > vert_prop;
typedef property<edge_weight_t, unsigned int> edge_prop;
```

```
typedef adjacency_list<listS, vecS, undirectedS, vert_prop,
  edge_prop> mygraph_t;
typedef mygraph_t::vertex_descriptor vertex_t;
```

Used in part 47a.

In order to implicitly create a graph, we specify a visitor that adds new vertices and edges to the graph when a vertex is examined (i.e., it adds children of that vertex). This visitor also records the sequence in which vertices are visited.

⟨Define visitor that generates a random tree on the fly 48a⟩ ≡

```
template <class VisitorType, class Sequence>
class implicit_visitor
  : public examine_recorder<VisitorType, Sequence>
{
public:
  implicit_visitor(Sequence& s)
    : examine_recorder<VisitorType, Sequence>(s),
      m_gen(time(0)), m_wgen(m_gen, 0, 9) {}
  template <class Vertex, class Graph>
  void examine_vertex(Vertex u, Graph& g) {
    examine_recorder<VisitorType, Sequence>::examine_vertex(u, g);
    if(num_v <= cur_v)
      return;
    unsigned int n = m_wgen();
    for(; n > 0; --n, ++cur_v) {
      Vertex v = add_vertex(vert_prop(white_color), g);
      add_edge(u, v, edge_prop(m_wgen()), g);
    }
  }
private:
  mt19937 m_gen;
  uniform_smallint<mt19937, unsigned int> m_wgen;
};
```

Used in part 47a.

We then begin the main program.

⟨Read vertex count argument from the command line 48b⟩ ≡

```
if(argc > 1)
  num_v = atoi(argv[1]);
if(!num_v) {
  cout << "Usage: " << argv[0]
       << " [num nodes (d. 40)]"
       << endl;
  return 1;
}
```

Used in part 47a.

We must add a start vertex to our graph, since astar_search requires a start vertex argument.

⟨Declare a graph and add a single start vertex 49a⟩ ≡

```
mygraph_t g;
vertex_t start = add_vertex(vert_prop(white_color), g);
```

Used in part 47a.

We create an instance of the visitor that generates a graph on the fly, and tell it to record the sequence of examined vertices in a list. We then call astar_search, passing the internal property maps.

⟨Run astar_search on the implicit graph 49b⟩ ≡

```
list<vertex_t> iseq;
implicit_visitor<default_astar_visitor, list<vertex_t> > ivis(iseq);
astar_search(g, start, astar_heuristic<mygraph_t, unsigned int>(),
             visitor(ivis).color_map(get(vertex_color, g)).
              rank_map(get(vertex_rank, g)).
              distance_map(get(vertex_distance, g)));
```

Used in part 47a.

The implicit graph search has caused the graph to become completely specified in memory. We now run astar_search again, this time with a visitor that *only* records the sequence of vertex examinations.

⟨Run astar_search again on the explicit graph 49c⟩ ≡

```
// now we have the graph completely built; search again to test
// for sequence correctness
list<vertex_t> eseq;
examine_recorder<default_astar_visitor, list<vertex_t> > evis(eseq);
astar_search(g, start, astar_heuristic<mygraph_t, unsigned int>(),
             visitor(evis).color_map(get(vertex_color, g)).
             rank_map(get(vertex_rank, g)).
             distance_map(get(vertex_distance, g)));
```

Used in part 47a.

The two recorded sequences should be the same. We check this in order to verify that the implicit graph search was correct.

⟨Verify that examination sequences are the same 49d⟩ ≡

```
accept(iseq.size() == eseq.size());
list<vertex_t>::iterator i = iseq.begin(), j = eseq.begin();
for(; i != iseq.end(); ++i, ++j)
  accept(*i == *j);
```

Used in part 47a.

This program compiles and works properly, and gives a simple example of using astar_search with implicit graphs.

5.2 Performance Testing

We test the performance of astar_search in two ways:

- By measuring the actual time that elapses over a series of runs; and
- By counting the number of vertices A* expands when searching for some goal, before finding the goal.

5.2.1 Heuristics

Unfortunately the "true" performance of A* depends largely on the heuristic functions with which it is used. For nearly any problem, a heuristic function can be contrived which will limit the performance of A* to the worst case (think, for example, of a heuristic which gives "good" paths high estimated costs, and "bad" paths low estimated costs). However, given a good heuristic function, A* can dramatically outperform other graph search algorithms.

Part of the genericity of our A* implementation is to place no limits on how a heuristic function is devised. Nevertheless, we design our performance tests to show that the use of a *good* heuristic with A* improves performance over "blind" searching. To this end, we for the most part use simple, admissible, known heuristics.

5.2.2 Verifying $O((E+V)\log V)$

In performance testing, we must first verify that our A* implementation's performance varies according to the $O((E+V)\log V)$ bound discussed in Section 3.4. (In fact, in most cases we will expect the performance to be only slightly worse than $O(E+V)$ since the size of OPEN will rarely approach the number of vertices in the graph. Note, however, that the complexity of the heuristic function being used can affect our running time.) This is simple to do by measuring the performance with random graphs of varying size.

Here we use a zero-heuristic and allow A* to search the entire connected component of the start vertex. By varying the size of the graph being searched, we verify that the running time of A* increases approximately linearly with graph size. The outline of this test is as follows:

"test-astar-complexity.cpp" 51a ≡

```
#include "test-astar-timelist.hpp"
⟨Include header files for random graph tests 39a⟩

int main(int argc, char **argv)
{
  unsigned int num_v, num_e;

  ⟨Declare a graph type for random graph tests 40b⟩

  for(num_v = 1000, num_e = 2 * num_v;
      num_v < 20000; num_v += 1000, num_e = 2 * num_v)
  {
    ⟨Generate a random graph 41a⟩
    ⟨Pick a random start vertex 41b⟩
    ⟨Randomly assign weights to edges 41c⟩

    ⟨Set up timing and find the median search time 51b⟩
  }
  return 0;
}
```

The only code new in this test is that for timing several runs of astar_search on the same graph, and keeping the median. We write the median time for each graph size to standard output (for use with gnuplot).

⟨Set up timing and find the median search time 51b⟩ ≡

```
mytimer_t t;
timelist_t<mytimer_t::value_type> times;
for(int i = 0; i < 20; ++i) {
  t.start();
  astar_search(g, start, astar_heuristic<mygraph_t, unsigned int>(),
               visitor(default_astar_visitor()));
  t.stop();
  times.record(t.time());
}
cout << num_v << "\t" << times.median() << endl;
```

Used in part 51a.

This test clearly shows that the running time of astar_search varies approximately linearly with the size of the graph (see Figure 5.2.2).

5.2.3 A* vs. BFS

In its most general form, the performance of our A* algorithm in visiting all nodes in a graph is not particularly meaningful. What *is* meaningful is the

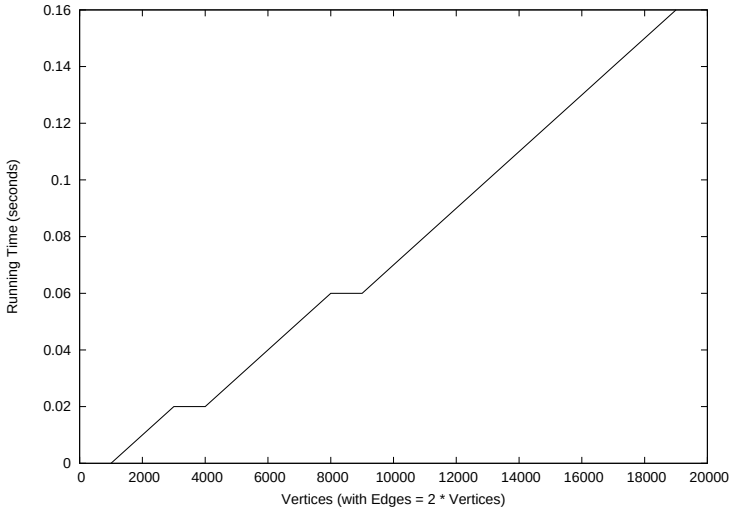

Figure 2: Running time of A* over a series of graph sizes.

amount of time it takes the algorithm to reach some goal state that is the "target" of the heuristic being used. In this sense, we can compare the performance of A* with that of BFS (which is also guaranteed to return the shortest path to a goal node for an unweighted graph), in terms of the number of vertices the two algorithms expand, and also in terms of pure running time.

In order to compare A* and BFS, we once again turn to the 8-puzzle. We continue using the Manhattan Distance heuristic from Section 1.4. Here, we compare the performance of A* with this heuristic to the performance of a "blind" search (BFS), in terms of the number of vertices examined in the search.

The code for the A* version of this test is completely specified in Section 1.4. The BFS code is nearly identical:

`"test-bfs-8puzzle.cpp"` 52 ≡

```
⟨Include files for 8-puzzle example 13a⟩
⟨Define a class to represent a puzzle board state 13b⟩
⟨Define a function for generating the successors of a state 14⟩
⟨Define graph and related types 15⟩
⟨Define a visitor that generates puzzle states on the fly 16⟩

int main(int argc, char **argv)
{
  ⟨Set up start and goal puzzle states 18a⟩
  try {
    ⟨Call BFS to search for the goal puzzle state 53⟩
```

```
  } catch(found_goal fg) {
    cout << "Number of vertices examined: "
         << examine_seq.size() << endl;
    return 0;
  }
  accept(false);
  return 0;
}
```

We need not define a heuristic, since BFS can't use one. The call to BFS is simple; we just set up a visitor and call the named parameter interface:

⟨Call BFS to search for the goal puzzle state 53⟩ ≡

```
puz_visitor<default_bfs_visitor> vis(psgoal, examine_seq);
breadth_first_search(g, start, visitor(vis).
                     color_map(get(vertex_color, g)));
```

Used in part 52.

As we expect, the performance of A* (with a Manhattan Distance heuristic) and BFS (completely blind) is hardly even comparable. We present three of the test instances we've tried (borrowed from [4]). The first test is very simple: the shortest path from the start state to the goal is only four moves. The start and goal board states are:

1	3	4		1	2	3
8		2	⟶	8		4
7	6	5		7	6	5

(Try solving this yourself. It should be very simple.) For this test, A* needs to examine only five states in order to find the optimal four-step solution. BFS, on the other hand, requires 27 state examinations to find the same solution!

The second test is only mildly more complicated—the solution is five steps long:

2	8	3		1	2	3
1	6	4	⟶	8		4
7		5		7	6	5

For this example, A* finds the shortest path after examining six states, compared with BFS's 58.

Our final test is much more difficult:

2	1	6		1	2	3
4		8	⟶	8		4
7	5	3		7	6	5

The optimal path for this test is 18 steps long. A* with the Manhattan distance heuristic finds the path after examining 30 states. BFS, on the other hand, cannot find the solution in reasonable time! A simple calculation ($4^{18} = 68,719,476,736$) shows that the worst-case number of states BFS might need to examine in order to find the goal is extremely large.

These examples give a clear picture of how the use of heuristics (problem-specific knowledge) can make a huge difference in reducing search times.

A Miscellaneous Source Code

Here we include miscellaneous source code for our A* implementation, and for testing of the implementation.

A.1 astar_search.hpp

The layout of the header file in which the A* implementation is specified is as follows:

"astar_search.hpp" 54 ≡

```
#ifndef BOOST_GRAPH_ASTAR_HPP
#define BOOST_GRAPH_ASTAR_HPP

⟨Include files needed for A* implementation 55a⟩

namespace boost {

  ⟨Concept check for AStarHeuristic concept 36a⟩
  ⟨Base astar_heuristic class 36b⟩

  ⟨Concept check for AStarVisitor concept 36c⟩
  ⟨Base astar_visitor class 37⟩

  namespace detail {
    ⟨Define BFS visitor that implements A* 33b⟩
  } // namespace detail

  ⟨Define non-named parameter interface to astar_search 30⟩
  ⟨Define named parameter interface to astar_search 32⟩
} // namespace boost

#endif // BOOST_GRAPH_ASTAR_HPP
```

The A* search implementation requires the inclusion of several headers, as follows.

⟨Include files needed for A* implementation 55a⟩ ≡

```
#include <functional>
#include <boost/limits.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/pending/mutable_queue.hpp>
#include <boost/graph/relax.hpp>
#include <boost/pending/indirect_cmp.hpp>
#include <boost/graph/exception.hpp>
#include <boost/graph/breadth_first_search.hpp>
```

Used in part 54.

Most of the important code for the A* implementation is specified in Section 4. Included here for completeness is miscellaneous code for the named parameter interface implementation, as well as code for the BFS visitor that makes BGL's BFS implementation perform its search according to the A* algorithm.

A.1.1 Named parameter interface details

Here we specify the "detail" code for the named parameter interface to the astar_search function. This code is responsible for setting up defaults for parameters that are not specified by name in the parameter list. The astar_dispatch1 function is called directly by the named parameter version of astar_search. The astar_dispatch2 function simply calls the non-named parameter interface after all necessary parameters have been set up.

⟨Detail code for named parameter interface to astar_search 55b⟩ ≡

```
namespace detail {
  template <class VertexListGraph, class AStarHeuristic,
            class CostMap, class DistanceMap, class WeightMap,
            class IndexMap, class ColorMap, class Params>
  inline void
  astar_dispatch2
    (VertexListGraph& g,
     typename graph_traits<VertexListGraph>::vertex_descriptor s,
     AStarHeuristic h, CostMap cost, DistanceMap distance,
     WeightMap weight, IndexMap index_map, ColorMap color,
     const Params& params)
  {
    dummy_property_map p_map;
    typedef typename property_traits<CostMap>::value_type C;
    astar_search
      (g, s, h,
       choose_param(get_param(params, graph_visitor),
                    make_astar_visitor(null_visitor())),
       choose_param(get_param(params, vertex_predecessor), p_map),
       cost, distance, weight, index_map, color,
```

```
        choose_param(get_param(params, distance_compare_t()),
                     std::less<C>()),
        choose_param(get_param(params, distance_combine_t()),
                     closed_plus<C>()),
        choose_param(get_param(params, distance_inf_t()),
                     std::numeric_limits<C>::max()),
        choose_param(get_param(params, distance_zero_t()),
                     C()));
    }

    template <class VertexListGraph, class AStarHeuristic,
              class CostMap, class DistanceMap, class WeightMap,
              class IndexMap, class ColorMap, class Params>
    inline void
    astar_dispatch1
      (VertexListGraph& g,
       typename graph_traits<VertexListGraph>::vertex_descriptor s,
       AStarHeuristic h, CostMap cost, DistanceMap distance,
       WeightMap weight, IndexMap index_map, ColorMap color,
       const Params& params)
    {
      typedef typename property_traits<WeightMap>::value_type D;
      typename std::vector<D>::size_type
        n = is_default_param(distance) ? num_vertices(g) : 1;
      std::vector<D> distance_map(n);
      n = is_default_param(cost) ? num_vertices(g) : 1;
      std::vector<D> cost_map(n);
      std::vector<default_color_type> color_map(num_vertices(g));
      default_color_type c = white_color;

      detail::astar_dispatch2
        (g, s, h,
         choose_param(cost, make_iterator_property_map
                      (cost_map.begin(), index_map,
                       cost_map[0])),
         choose_param(distance, make_iterator_property_map
                      (distance_map.begin(), index_map,
                       distance_map[0])),
         weight, index_map,
         choose_param(color, make_iterator_property_map
                      (color_map.begin(), index_map, c)),
         params);
    }
  } // namespace detail
```

Used in part 32.

A.1.2 A* BFS visitor details

A special BFS visitor modifies the breadth-first behavior of BGL's BFS, changing the search to follow A*'s rules for choosing the order in which to examine vertices. The details of this implementation are as follows:

⟨BFS visitor type definitions 57a⟩ ≡

```
typedef typename property_traits<CostMap>::value_type C;
typedef typename property_traits<ColorMap>::value_type ColorValue;
typedef color_traits<ColorValue> Color;
```

Used in part 33b.

The visitor constructor initializes all of the member variables of the visitor:

⟨BFS visitor constructor 57b⟩ ≡

```
astar_bfs_visitor(AStarHeuristic h, UniformCostVisitor vis,
                  UpdatableQueue& Q, PredecessorMap p,
                  CostMap c, DistanceMap d, WeightMap w,
                  ColorMap col, BinaryFunction combine,
                  BinaryPredicate compare, C zero)
  : m_h(h), m_vis(vis), m_Q(Q), m_predecessor(p), m_cost(c),
    m_distance(d), m_weight(w), m_color(col),
    m_combine(combine), m_compare(compare), m_zero(zero) {}
```

Used in part 33b.

Most of the visitor events do nothing but call the corresponding event for the AStarVisitor passed by the user to astar_search. These "passthru" events are specified as such:

⟨Basic (mostly "passthru") events 57c⟩ ≡

```
template <class Vertex, class Graph>
void initialize_vertex(Vertex u, Graph& g) {
  m_vis.initialize_vertex(u, g);
}
template <class Vertex, class Graph>
void discover_vertex(Vertex u, Graph& g) {
  m_vis.discover_vertex(u, g);
}
template <class Vertex, class Graph>
void examine_vertex(Vertex u, Graph& g) {
  m_vis.examine_vertex(u, g);
}
template <class Vertex, class Graph>
void finish_vertex(Vertex u, Graph& g) {
  m_vis.finish_vertex(u, g);
```

```
}
template <class Edge, class Graph>
void examine_edge(Edge e, Graph& g) {
  if (m_compare(get(m_weight, e), m_zero))
    throw negative_edge();
  m_vis.examine_edge(e, g);
}
template <class Edge, class Graph>
void non_tree_edge(Edge, Graph&) {}
```

Used in part 33b.

Note that the last two functions are slightly different. The examine_edge event enforces the requirement that edge weights in the graph being searched be non-negative. Finally, we declare the member variables for the BFS visitor, mostly property maps, along with the heuristic, visitor and cost manipulation functions (compare, combine) specified by the user's call to astar_search.

⟨BFS visitor member variables 58a⟩ ≡

```
AStarHeuristic m_h;
UniformCostVisitor m_vis;
UpdatableQueue& m_Q;
PredecessorMap m_predecessor;
CostMap m_cost;
DistanceMap m_distance;
WeightMap m_weight;
ColorMap m_color;
BinaryFunction m_combine;
BinaryPredicate m_compare;
bool m_decreased;
C m_zero;
```

Used in part 33b.

A.2 Visitors for testing

Our correctness and performance tests made use of several visitors for which it was useful to provide common implementations.

"test-astar-visitors.hpp" 58b ≡

```
#ifndef _TEST_ASTAR_VISITORS_HPP
#define _TEST_ASTAR_VISITORS_HPP

#include <astar_search.hpp>
#include <iostream>
```

⟨Define a generic visitor for recording vertex examinations 59a⟩

⟨Define a generic visitor that prints on vertex examination 59b⟩

⟨Define a visitor that prints every event 59c⟩

```
#endif // _TEST_ASTAR_VISITORS_HPP
```

We record vertex examinations in a templated Sequence container with a visitor that can be derived from any other visitor type (so that it can also be used, for example, with BFS).

⟨Define a generic visitor for recording vertex examinations 59a⟩ ≡

```
template <class VisitorType, class Sequence>
class examine_recorder : public VisitorType
{
public:
  examine_recorder(Sequence& s)
    : m_seq(s) {}
  template <class Vertex, class Graph>
  void examine_vertex(Vertex u, Graph& g) {
    m_seq.push_back(u);
    VisitorType::examine_vertex(u, g);
  }
protected:
  Sequence& m_seq;
};
```

Used in part 58b.

In some cases we just want to print the sequence as it happens, rather than record it.

⟨Define a generic visitor that prints on vertex examination 59b⟩ ≡

```
template <class VisitorType>
class print_examine_visitor : public VisitorType
{
public:
  template <class Vertex, class Graph>
  void examine_vertex(Vertex u, Graph& g) {
    std::cout << u << std::endl;
    VisitorType::examine_vertex(u, g);
  }
};
```

Used in part 58b.

In a similar vein, it is often useful to print the entire sequence of events.

⟨Define a visitor that prints every event 59c⟩ ≡

```
class astar_print_visitor : public boost::default_astar_visitor
{
public:
  template <class Vertex, class Graph>
  void initialize_vertex(Vertex u, Graph& g) {
    std::cout << "initialize_vertex(" << u << ")" << std::endl;
  }

  template <class Vertex, class Graph>
  void discover_vertex(Vertex u, Graph& g) {
    std::cout << "discover_vertex(" << u << ")" << std::endl;
  }

  template <class Vertex, class Graph>
  void examine_vertex(Vertex u, Graph& g) {
    std::cout << "examine_vertex(" << u << ")" << std::endl;
  }

  template <class Edge, class Graph>
  void examine_edge(Edge e, Graph& g) {
    std::cout << "examine_edge( (" << boost::source(e, g) << ","
              << boost::target(e, g) << ") )" << std::endl;
  }

  template <class Edge, class Graph>
  void edge_relaxed(Edge e, Graph& g) {
    std::cout << "edge_relaxed( (" << boost::source(e, g) << ","
              << boost::target(e, g) << ") )" << std::endl;
  }

  template <class Edge, class Graph>
  void edge_not_relaxed(Edge e, Graph& g) {
    std::cout << "edge_not_relaxed( (" << boost::source(e, g) << ","
              << boost::target(e, g) << ") )" << std::endl;
  }

  template <class Edge, class Graph>
  void black_target(Edge e, Graph& g) {
    std::cout << "black_target( (" << boost::source(e, g) << ","
              << boost::target(e, g) << ") )" << std::endl;
  }

  template <class Vertex, class Graph>
  void finish_vertex(Vertex u, Graph& g) {
    std::cout << "finish_vertex(" << u << ")" << std::endl;
  }
};
```

Used in part 58b.

A.3 Alternate A*

Included here is a separate implementation of A*, adapted from that in [2]. This implementation is also generic (though less so than our BGL implementation). Since we've used this implementation in our correctness testing, the code is included here. This implementation was chosen for its simplicity and ease of use—it is not comparable to our implementation in terms of performance.

"alternate_astar.hpp" 61a ≡

```
#ifndef _ALTERNATE_ASTAR_HPP
#define _ALTERNATE_ASTAR_HPP

#include <list>
#include <algorithm>
#include <limits>

typedef unsigned int cost_t;
const cost_t SEARCH_MAXCOST =
  std::numeric_limits<cost_t>::max();
```

⟨Define a searchNode class for storing vertex information 61b⟩

⟨Implement alternate A* 62⟩

```
#endif // _ALTERNATE_ASTAR_HPP
```

Rather than use property maps like BGL, this A* stores information about each vertex, such as predecessor and f-value, in a "searchNode" structure, defined as follows:

⟨Define a searchNode class for storing vertex information 61b⟩ ≡

```
template<class T,class S>
class searchNode
{
public:
  const T node;
  const searchNode<T,S> *parent;
  cost_t g, h, f;

  searchNode(const T &n, cost_t c, const searchNode<T,S> *p,
    const cost_t &pdist)
    : node(n), parent(p), g(pdist), h(c)
  {
    if( p )
      g += p->g;
    f = g + h;
  }
  inline bool operator==(const T &n) const { return node == n; }
```

```
  inline bool operator==(const searchNode<T,S> &n) const
  {
    return node == n.node;
  }
};
```

Used in part 61a.

The A* implementation itself closely follows Algorithm 1, though it does not take a goal argument and terminates only when all vertices in the same component as the start have been examined. It requires its first argument (the "graph") to provide functions for calculating the heuristic value of a vertex, and for providing the neighbors of a vertex. We have also modified the function to write the shortest-path distances to a distance map after the algorithm finishes.

⟨Implement alternate A* 62⟩ ≡

```
template<typename T, typename S, typename DistanceMap>
void graphSearch(S &graph, T &start, DistanceMap &dmap)
{
  std::list< searchNode<T,S> > open, closed;
  typename std::list< searchNode<T,S> >::iterator i, minnode, k;
  std::list< std::pair<T, cost_t> > neighbors;
  typename std::list< std::pair<T, cost_t> >::iterator m;
  cost_t mincost;

  open.push_back(searchNode<T,S>(start,0,0,0));
  while(open.size()) {
    mincost = SEARCH_MAXCOST;
    // find the lowest cost on open
    for(i = open.begin(); i != open.end(); i++) {
      if(i->f < mincost) {
        mincost = i->f;
        minnode = i;
      }
    }
    // move min cost node from open to closed
    searchNode<T,S> tmp(*minnode);
    open.erase(minnode);
    closed.push_back(tmp);

    // now, get each neighbor of the minimum cost node
    neighbors.clear();
    graph.getNodeNeighbors(tmp.node, neighbors);
    for(m = neighbors.begin(); m != neighbors.end(); m++) {
      searchNode<T,S> newNode(m->first, graph.getNodeCost(m->first),
        &(closed.back()), m->second);
      k = std::find(open.begin(), open.end(), newNode);
      if(k != open.end()) {
```

```
        if(k->g > newNode.g) {
          // found a better path to this node
          open.erase(k);
          open.push_back(newNode);
        }
      } else {
        k = std::find(closed.begin(), closed.end(), newNode);
        if(k == closed.end())
          open.push_back(newNode);
        else if(k->g > newNode.g) {
          // found a better path to this node
          closed.erase(k);
          open.push_back(newNode);
        }
      }
    }
  }
  for(i = closed.begin(); i != closed.end(); ++i)
    dmap[i->node] = i->g;
}
```

Used in part 61a.

A.4 Timing

Our performance tests require the ability to time algorithm executions, and to record a list of times (and find the median of this list). Similarly, we sometimes need to record a list of *examination counts* (the number of vertices examined by the algorithm) and find the median of this list. We provide a very simple generic container, based on code from Musser, Derge and Saini [5]. We also provide a simple mechanism for computing the running time of a single execution.

"test-astar-timelist.hpp" 63a ≡

```
#ifndef _TEST_ASTAR_TIMELIST_HPP
#define _TEST_ASTAR_TIMELIST_HPP

#include <vector>
#include <ctime>

⟨Define a container for recording times or examination counts 63b⟩
⟨Define a class for computing running time of one execution 64a⟩

#endif // _TEST_ASTAR_TIMELIST_HPP
```

Our container is based on an STL vector:

⟨Define a container for recording times or examination counts 63b⟩ ≡

```
template <class T>
class timelist_t : public std::vector<T>
{
public:
  void record(T v) { push_back(v); }
  T median() {
    typename timelist_t<T>::iterator m = begin() + (end() - begin()) / 2;
    std::nth_element(begin(), m, end());
    return *m;
  }
};
```

Used in part 63a.

In order to compute the running time of a single execution, a program records the time immediately prior to the algorithm call, and the time immediately after the call returns, and then computes their difference.

⟨Define a class for computing running time of one execution 64a⟩ ≡

```
class mytimer_t
{
public:
  typedef double value_type;
  void start() { a = clock(); }
  void stop() { b = clock(); }
  double time() const { return (b - a) / CLOCKS_PER_SEC; }
protected:
  double a, b;
};
```

Used in part 63a.

A.5 Acceptance testing

Our acceptance testing uses a simple utility (really just a modification of the standard assert function) that outputs the text "FAILURE" to standard output, for easy parsing by scripts.

"test-astar-accept.hpp" 64b ≡

```
#ifndef _TEST_ASTAR_ACCEPT_HPP
#define _TEST_ASTAR_ACCEPT_HPP

#include <iostream>
#include <stdlib.h>

#define accept(b) accept_fail(b, __FILE__, __LINE__, \
```

```
                                __STRING(b))

bool accept_fail(bool b, const char *file, int line,
                 const char *expr)
{
  if(!b) {
    std::cout << "FAILURE: " << file << ":" << line << " "
              << expr << std::endl;
    abort();
  }
  return b;
}

#endif // _TEST_ASTAR_ACCEPT_HPP
```

A.6 BFS for implicit graphs

In order to use astar_search with implicit graphs, it was necessary to slightly modify BGL's breadth-first search implementation (upon which astar_search is based). For more information, see Section 1.4. For reasons of brevity, we do not display the code here. It is included in the file nonconst_bfs.hpp.

A.7 Makefile

The following Makefile builds our tests and documentation.

"Makefile" 65 ≡

```
DOCNAME = astar-bgl
SOURCES = astar_search.hpp nonconst_bfs.hpp test-astar-accept.hpp \
        test-astar-bfs.cpp test-astar-cities.cpp \
        test-astar-complexity.cpp test-astar-complexity.plot \
        test-astar-dijkstra.cpp test-astar-implicit.cpp \
        test-astar-list.cpp test-astar-alternate.cpp \
        test-astar-timelist.hpp test-astar-visitors.hpp \
        alternate_astar.hpp test-astar-8puzzle.cpp \
        test-bfs-8puzzle.cpp

all: doc

srcclean:
        -rm -f $(DOCNAME).tex $(SOURCES)

#
# building documentation
#

doc: docps docpdf
```

```
docps:
	nuweb $(DOCNAME)
	latex $(DOCNAME)
	nuweb $(DOCNAME)
	latex $(DOCNAME)
	dvips $(DOCNAME) -o

docpdf:
	pdfnuweb $(DOCNAME)
	pdflatex $(DOCNAME)
	pdfnuweb $(DOCNAME)
	pdflatex $(DOCNAME)

docclean:
	-rm -f $(DOCNAME).ps $(DOCNAME).pdf
	-rm -f $(DOCNAME).dvi $(DOCNAME).aux $(DOCNAME).log $(DOCNAME).loa
	-rm -f $(DOCNAME).brf $(DOCNAME).out $(DOCNAME).toc

#
# building test programs
#

ASTARDIR = .
BOOSTDIR = /cs/musser/public.html/gsd/boost-1.30.2
CXX = g++33
#BOOSTDIR = /home/kris/rpi/generic/boost-1.30.2
#CXX = g++-3.2

CC = $(CXX)
LD = $(CXX)
CXXFLAGS += -O3 -Wall -ftemplate-depth-30 -I$(BOOSTDIR) -I$(ASTARDIR)
LDFLAGS += -lstdc++

TESTS = test-astar-alternate test-astar-cities test-astar-list \
	test-astar-implicit test-astar-bfs test-astar-dijkstra \
	test-astar-complexity test-astar-8puzzle test-bfs-8puzzle

test: $(TESTS)

runtests: test
	sh run-test-suite.sh

figures: test
	-neato -s -n -Tps test-astar-cities.dot -o test-astar-cities.ps
	-ps2epsi test-astar-cities.ps test-astar-cities.eps
	-epstopdf test-astar-cities.eps
	-gnuplot test-astar-complexity.plot
	-epstopdf test-astar-complexity.eps
```

```
testclean:
        -rm -f $(TESTS) *.o
        -rm -f test-astar-cities.{dot,ps}
        -rm -f test-astar-alternate.{dot,ps}
        -rm -f test-astar-complexity.dat
#
# etc.
#

dist:
        mkdir astar-bgl; \
        cp astar-bgl.w test-astar-complexity.{eps,pdf} \
                test-astar-cities.{eps,pdf} astar-bgl; \
        tar cfz astar-bgl.tar.gz astar-bgl; \
        rm -rf astar-bgl

clean: testclean docclean srcclean
        -rm -f run-test-suite.sh
        -rm -f astar-bgl.tar.gz
        -rm -f *~

reallyclean: clean
        -rm -f test-astar-cities.{eps,pdf}
        -rm -f test-astar-complexity.{eps,pdf}
        -rm -f Makefile
```

References

[1] The Boost Graph Library, 2003, http://www.boost.org. 1.2

[2] K. Beevers. Generic A* implementation for robot motion planners. 2002. Available upon request. 5.1.1, A.3

[3] K. Beevers. Heuristic search methods (lecture notes). http://www.cs.rpi.edu/~beevek/files/ai-heuristic-search.pdf, September 2003. 1.1, 3.2

[4] Justin Heyes-Jones. A* algorithm tutorial. http://www.geocities.com/jheyesjones/astar.html, September 2001. 1.1, 5.2.3

[5] D.R. Musser, G.J. Derge and A. Saini. *STL Tutorial and Reference Guide, Second edition*. Addison-Wesley, 2001. A.4

[6] N.J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kauffmann, San Fransisco, CA, 1998. 1.1

[7] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, Second edition*. Prentice Hall, 2003. 1.1

[8] J.G. Siek, L.-Q. Lee and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002. 1.2, 1.4