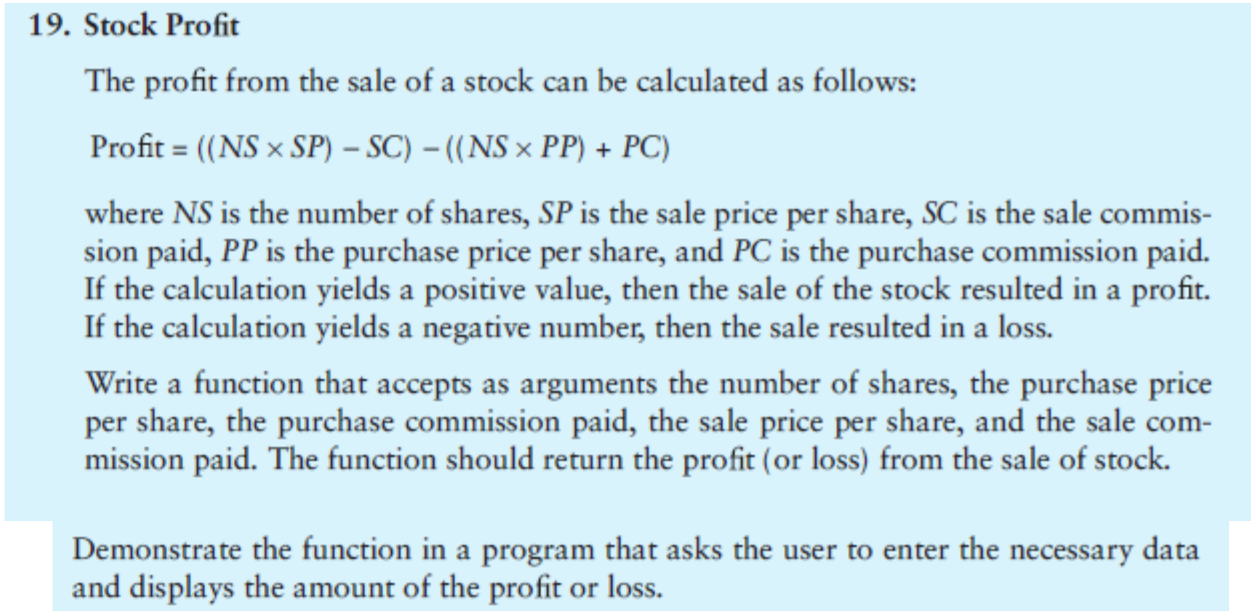
C++ is Fun – Part Seven

at Turbine/Warner Bros.!

Russell Hanson

Homework #3 Exercises (pick 2)

1)



19. Stock Profit

The profit from the sale of a stock can be calculated as follows:

$$\text{Profit} = ((NS \times SP) - SC) - ((NS \times PP) + PC)$$

where NS is the number of shares, SP is the sale price per share, SC is the sale commission paid, PP is the purchase price per share, and PC is the purchase commission paid. If the calculation yields a positive value, then the sale of the stock resulted in a profit. If the calculation yields a negative number, then the sale resulted in a loss.

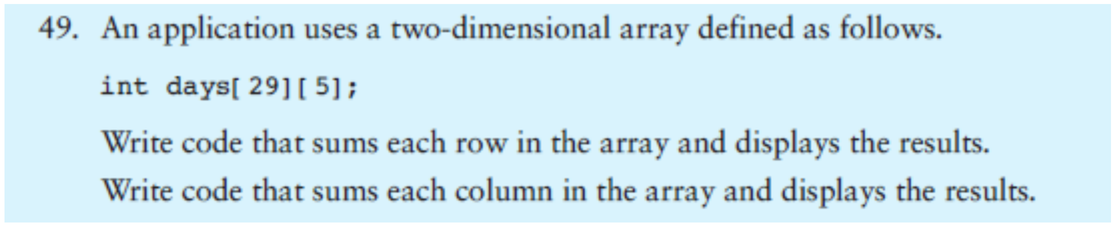
Write a function that accepts as arguments the number of shares, the purchase price per share, the purchase commission paid, the sale price per share, and the sale commission paid. The function should return the profit (or loss) from the sale of stock.

Demonstrate the function in a program that asks the user to enter the necessary data and displays the amount of the profit or loss.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
float myfunction (float ns, float sp, float sc, float pp, float pc) {
   return (((ns*sp)-sc)-((ns*pp)+pc));
}

int _tmain(int argc, _TCHAR* argv[]){
   float ns=100, sp=45.95, sc=8.00, pp=6.95, pc=8.00;
   cout << "Enter Number of Shares, Sale Price Per Share, Sale Commission, Purchase Price per Share, Purchase Commission Paid,\n";
   cout << "separated by spaces (\' \'):";
   cin >> ns >> sp >> sc >> pp >> pc;
   cout << "\nThe profit of this transaction is $" << myfunction(ns, sp, sc, pp, pc) << endl << endl;

   system("Pause");
   return 0;
}
```

```
//
// stock profit
//   PROFIT = ((NS x SP) - SC) - ((NS x PP) + PC)
//                                              NS = number of shares
//                                              SP = sale price per share
//                                              SC = sale commision paid
//                                              PP purchase price per share
//                                              PC purchase commission paid
// if ( PROFIT > 0 )
//                          sale results in profit
// if ( PROFIT < 0 )
//    sale results in loss
//
#include "stdafx.h"
#include <iostream>
#include <string>

using namespace std;

// prototypes
double stockProfit(int, double, double, double, double);
int menuGetInt(std::string);
double menuGetDouble(std::string);

double stockProfit(int numShares, double purchasePricePerShare, double
purchaseCommissionPaid, double salePricePerShare, double saleCommissionPaid)
{
                        return ((numShares*salePricePerShare)-saleCommissionPaid)-
((numShares*purchasePricePerShare)+purchaseCommissionPaid);
}

int menuGetInt(std::string prompt) {
                        int result = 0;
                        do
                        {
                                                cout << prompt << endl;
                                                string line;
                                                getline( cin, line );
                                                result = atoi(line.c_str());
                                                if ( result == 0 )
                                                                        cout << "Invalid entry: " << line << endl;
                        } while (result == 0);
                        return result;
}

double menuGetDouble(std::string prompt) {
                        double result = 0;
                        do
                        {
                                                cout << prompt << endl;
                                                string line;
                                                getline( cin, line );
                                                result = atof(line.c_str());
                                                if ( result == 0.0 )
                                                                        cout << "Invalid entry: " << line << endl;
                        } while (result == 0.0);
                        return result;
}

int main(int argc, char* argv[])
{
                        int numShares = menuGetInt("Number of shares: ");
                        double purchasePricePerShare = menuGetDouble("Purchase Price per Share: ");
                        double purchaseCommissionPaid = menuGetDouble("Purchase
Commission Paid: ");
                        double salePricePerShare = menuGetDouble("Sale Price Per Share: ");
                        double saleCommissionPaid = menuGetDouble("Sale Commission Paid: ");;
                        double profit = stockProfit(numShares, purchasePricePerShare,
                                                purchaseCommissionPaid, salePricePerShare,
saleCommissionPaid);

                        cout << endl;
                        if ( profit > 0 )
                                                cout << "Profit: $" << profit << endl;
                        else if ( profit < 0 )
                                                cout << "Loss: $" << profit*-1 << endl;
                        else
                                                cout << "No profit or loss" << endl;
                        cout << endl;

                        system("PAUSE");
                        return 0;
}
```

2)



49. An application uses a two-dimensional array defined as follows.

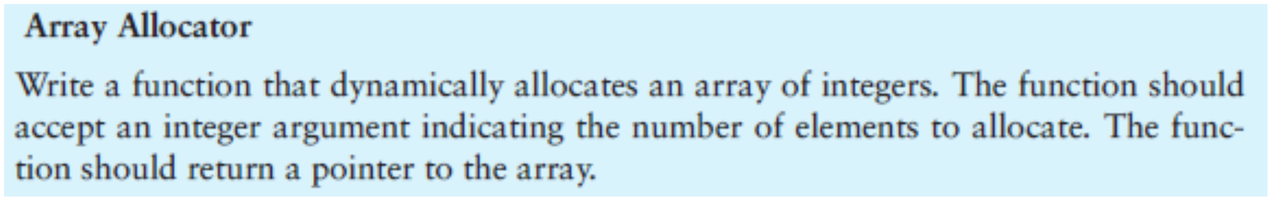
```
int days[29][5];
```

Write code that sums each row in the array and displays the results.

Write code that sums each column in the array and displays the results.

```
#include "stdafx.h"
#include <ctime>
#include <cstdlib>
#include <iostream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   int days[29][5];
   srand(time(0));
   for (int i=0; i<29; i++)
      for (int j=0; j<5; j++)
         days[i][j]=rand()%10000;
   int rows=0, columns=0;
   for (int i=0; i<29; i++) {
      for (int j=0; j<5; j++)
         columns +=days[i][j];
      cout << "Sum of column " << i <<" is "<< columns << endl;
      columns = 0;
   }
   for (int j=0; j<5; j++) {
      for (int i=0; i<29; i++)
         rows +=days[i][j];
      cout << "Sum of row " << j <<" is "<< rows << endl;
      rows = 0;

   }
   system("Pause");
   return 0;
}
```

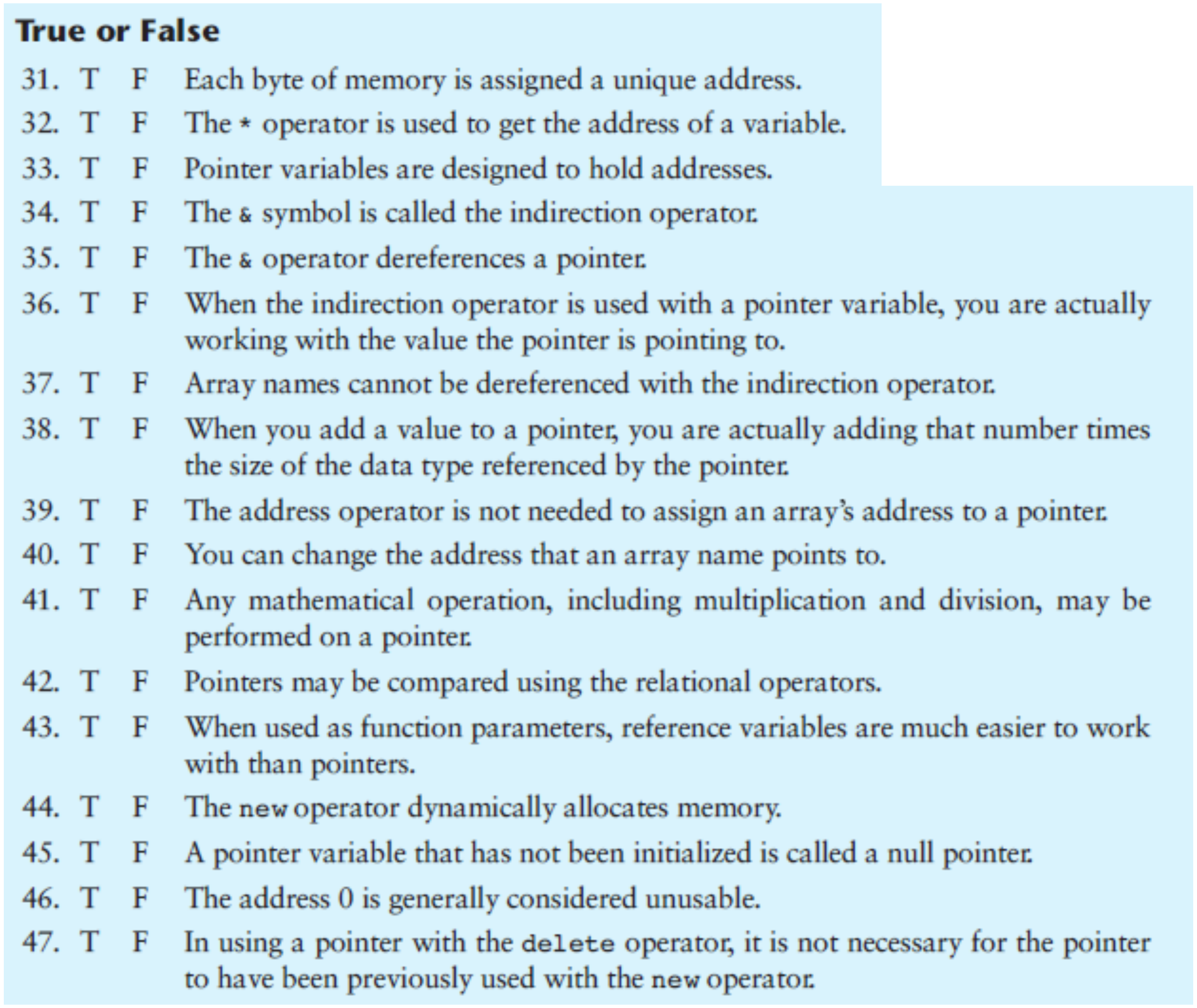
3)



Array Allocator

Write a function that dynamically allocates an array of integers. The function should accept an integer argument indicating the number of elements to allocate. The function should return a pointer to the array.

4)



True or False

31. T F Each byte of memory is assigned a unique address.
32. T F The `*` operator is used to get the address of a variable.
33. T F Pointer variables are designed to hold addresses.
34. T F The `&` symbol is called the indirection operator.
35. T F The `&` operator dereferences a pointer.
36. T F When the indirection operator is used with a pointer variable, you are actually working with the value the pointer is pointing to.
37. T F Array names cannot be dereferenced with the indirection operator.
38. T F When you add a value to a pointer, you are actually adding that number times the size of the data type referenced by the pointer.
39. T F The address operator is not needed to assign an array's address to a pointer.
40. T F You can change the address that an array name points to.
41. T F Any mathematical operation, including multiplication and division, may be performed on a pointer.
42. T F Pointers may be compared using the relational operators.
43. T F When used as function parameters, reference variables are much easier to work with than pointers.
44. T F The `new` operator dynamically allocates memory.
45. T F A pointer variable that has not been initialized is called a null pointer.
46. T F The address 0 is generally considered unusable.
47. T F In using a pointer with the `delete` operator, it is not necessary for the pointer to have been previously used with the `new` operator.

TRUE FALSE ANSWERS

31. TRUE
32. FALSE
33. TRUE
34. FALSE
35. FALSE
36. TRUE
37. FALSE
38. TRUE
39. TRUE
40. FALSE
41. FALSE
42. TRUE
43. TRUE
44. TRUE
45. FALSE
46. TRUE
47. FALSE

5) Implement three different pointer casts such as dynamic_cast, __try_cast, static_cast, reinterpret_cast, const_cast, C-Style Casts as described in http://msdn.microsoft.com/en-us/library/aa712854(v=vs.71).aspx

Using references vs. pointers

```
string concatenate(string first,  string second,  string* middle)
{
	stringstream ss;
	ss << first << " " << *middle << " " << second;
	string s = ss.str();
	*middle = "passed by reference”;
	return s;
}

int main()
{
	string firstWord;
	string secondWord;
	string middleWord;

	cout << "Please enter a word:" << endl;
	cin >> firstWord;

	cout << "Please enter a word to put in the middle:" << endl;
	cin >> middleWord;

	cout << "Please enter a third word:" << endl;
	cin >> secondWord;

	string phrase = concatenate(firstWord, secondWord,
&middleWord);

	cout << "Your final sentence is: " << phrase << endl;

	cout << ”Modified: " << middleWord;

	system("PAUSE");
}
```

```
string concatenate(const string& first, const string& second, const
string& middle)
{
			stringstream ss;

			ss << first << " " << middle << " " << second;
			string s = ss.str();

			return s;
}

int main()
{
			string firstWord;
			string secondWord;
			string middleWord;

			cout << "Please enter a word:" << endl;
			cin >> firstWord;

			cout << "Please enter a word to put in the middle:" << endl;
			cin >> middleWord;

			cout << "Please enter a third word:" << endl;
			cin >> secondWord;

			string phrase = concatenate(firstWord, secondWord, middleWord);
			cout << "Your final sentence is: " << phrase << endl;

			system("PAUSE");
}
```

```
class Sprite {
    ...
        bool checkCollision(Sprite &spr);
    ...
};
```

So, if I have that class, I can do this:

```
ball.checkCollision(bar1);
```

But if I change the class to this:

```
class Sprite {
    ...
        bool checkCollision(Sprite* spr);
    ...
};
```

I have to do this:

```
ball.checkCollision(&bar1);
```

So, what's the difference?? It's better a way instead other?

In both cases you are actually passing the address of *bar1* (and you're not copying the value), since both pointers (Sprite *) and references (Sprite &) have reference semantics, in the first case explicit (you have to explicitly dereference the pointer to manipulate the pointed object, and you have to explicitly pass the address of the object to a pointer parameter), in the second case implicit (when you manipulate a reference it's *as if* you're manipulating the object itself, so they have value syntax, and the caller's code doesn't explicitly pass a pointer using the & operator).

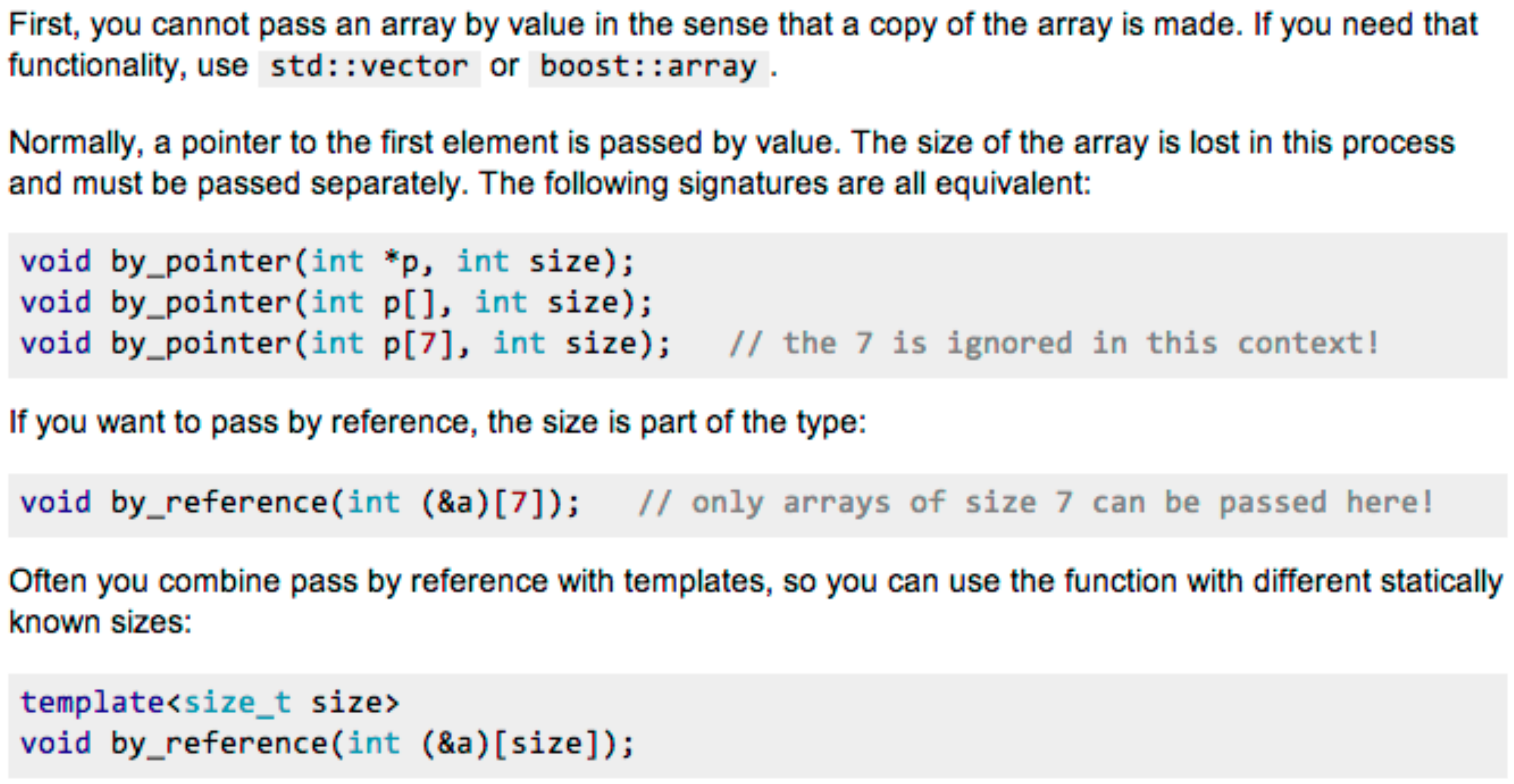
So, the big difference between pointers and references is on what you can do on the pointer/reference variable: pointer variables themselves can be modified, so they may be changed to point to something else, can be NULLed, incremented, decremented, etc, so there's a strong separation between activities on the pointer (that you access directly with the variable name) and on the object that it points to (that you access with the * operator - or, if you want to access to the members, with the -> shortcut).

References, instead, aim to be just an alias to the object they point to, and do not allow changes to the reference itself: you initialize them with the object they refer to, and then they act as if they were such object for their whole life.

In general, in C++ references are preferred over pointers, for the motivations I said and for some other that you can find in the appropriate section of C++ FAQ Lite.

In terms of performance, they should be the same, because a reference is actually a pointer in disguise; still, there may be some corner case in which the compiler may optimize more when the code uses a reference instead of a pointer, because references are guaranteed not to change the address they hide (i.e., from the beginning to the end of their life they always point to the same object), so in some strange case you *may* gain something in performance using references, but, again, the point of using references is about good programming style and readability, not performance.

Arrays are passed as pointers, not by value



First, you cannot pass an array by value in the sense that a copy of the array is made. If you need that functionality, use `std::vector` or `boost::array`.

Normally, a pointer to the first element is passed by value. The size of the array is lost in this process and must be passed separately. The following signatures are all equivalent:

```
void by_pointer(int *p, int size);
void by_pointer(int p[], int size);
void by_pointer(int p[7], int size);   // the 7 is ignored in this context!
```

If you want to pass by reference, the size is part of the type:

```
void by_reference(int (&a)[7]);   // only arrays of size 7 can be passed here!
```

Often you combine pass by reference with templates, so you can use the function with different statically known sizes:

```
template<size_t size>
void by_reference(int (&a)[size]);
```

```
// prototype(s)
void mysort(int[], int, bool); // equivalent to void mysort(int*, int, bool);
void display_array(int[], int);

int main(int argc, char *argv[])
{
        int testDesc[] = {6, 1, 2, 5, 3, 9};
        int items = sizeof(testDesc)/sizeof(int);

        display_array(testDesc, items);
        // sort descending by reference
        mysort(testDesc, items, true);
        display_array(testDesc, items);

        int testAsc[] = {1, 6, 10, 100, 200, 2, 5, 3, 9, 60};
        items = sizeof(testAsc)/sizeof(int);

        display_array(testAsc, items);
        // sort ascending by reference
        mysort(testAsc, items, false);
        display_array(testAsc, items);

        return 0;
}
```

```
void mysort(int list[], int size, bool desc)
{
        int match;
        int temp;

        for (int i = size - 1; i > 0; i--)
        {
        match = 0;

        for (int j = 1; j <= i; j++) {
                if (desc)
                {
                        if ( list[j] < list[match] )
                                match = j;
                }
                else
                {
                        if (list[j] > list[match])
                                match = j;
                }
        }

        // swap values
        temp = list[match];
        list[match] = list[i];
        list[i] = temp;
        }
}
```

11.10 Focus on Software Engineering: When to Use ., When to Use ->, and When to Use *

Sometimes structures contain pointers as members. For example, the following structure declaration has an `int` pointer member:

```
struct GradeInfo
{
   string name;                     // Student names
   int *testScores;                 // Dynamically allocated array
   float average;                   // Test average
};
```

It is important to remember that the structure pointer operator (->) is used to dereference a pointer to a structure, not a pointer that is a member of a structure. If a program dereferences the `testScores` pointer in this structure, the indirection operator must be used. For example, assume that the following variable has been defined:

```
GradeInfo student1;
```

The following statement will display the value pointed to by the `testScores` member:

```
cout << *student1.testScores;
```

It is still possible to define a pointer to a structure that contains a pointer member. For instance, the following statement defines `stPtr` as a pointer to a `GradeInfo` structure:

```
GradeInfo *stPtr;
```

Assuming that `stPtr` points to a valid `GradeInfo` variable, the following statement will display the value pointed to by its `testScores` member:

```
cout << *stPtr->testScores;
```

In this statement, the `*` operator dereferences `stPtr->testScores`, while the `->` operator dereferences `stPtr`. It might help to remember that the following expression:

```
stPtr->testScores
```

is equivalent to

```
(*stPtr).testScores
```

Table 11-3

Expression	Description
`s->m`	`s` is a structure pointer and `m` is a member. This expression accesses the `m` member of the structure pointed to by `s`.
`*a.p`	`a` is a structure variable and `p`, a pointer, is a member. This expression dereferences the value pointed to by `p`.
`(*s).m`	`s` is a structure pointer and `m` is a member. The `*` operator dereferences `s`, causing the expression to access the `m` member of the structure pointed to by `s`. This expression is the same as `s->m`.
`*s->p`	`s` is a structure pointer and `p`, a pointer, is a member of the structure pointed to by `s`. This expression accesses the value pointed to by `p`. (The `->` operator dereferences `s` and the `*` operator dereferences `p`.)
`*(*s).p`	`s` is a structure pointer and `p`, a pointer, is a member of the structure pointed to by `s`. This expression accesses the value pointed to by `p`. `(*s)` dereferences `s` and the outermost `*` operator dereferences `p`. The expression `*s->p` is equivalent.

Quite a few topics in AI…

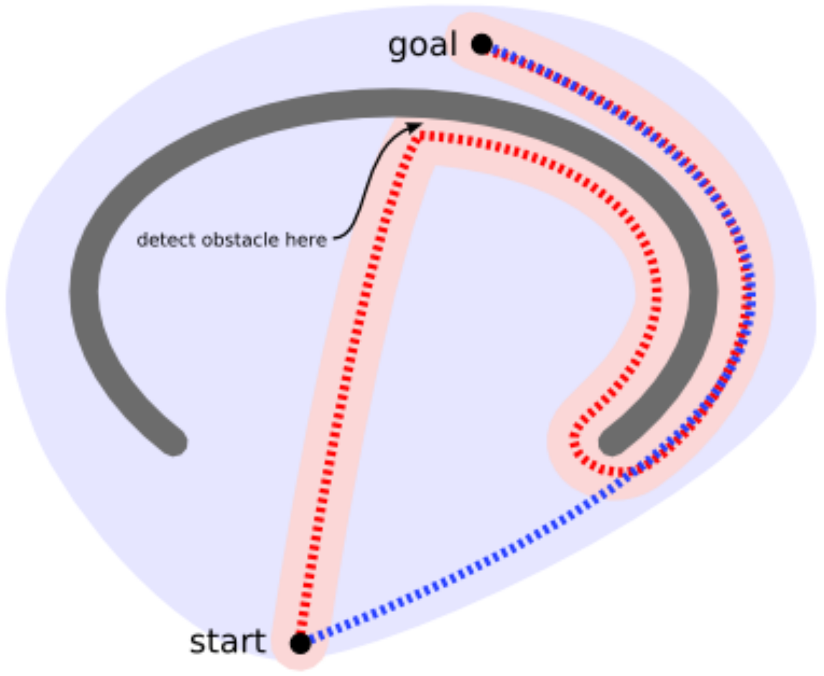
-genetic algorithms
-neural networks
-rules-based systems
-clustering algorithms
-ant algorithms
-fuzzy logic
-hidden Markov models
-simulated annealing
-intelligent agents

-classifier systems
-natural language processing
-particle swarm optimization
-A-Star pathfinding
-reinforcement learning

Applications include personalization engine, rules-based reasoning system, character trainer for game AI, Web-based news agent, genetic code optimizer, artificial life simulation, etc.

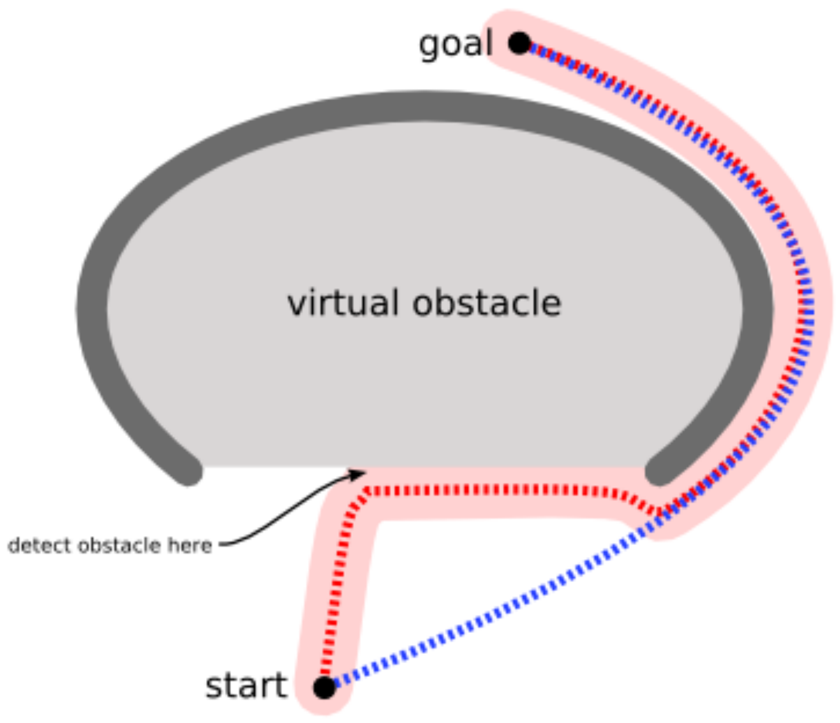
AI as pathfinding and planning

Movement for a single object seems easy. Pathfinding is complex. Why bother with pathfinding? Consider the following situation:



The unit is initially at the bottom of the map and wants to get to the top. There is nothing in the area it scans (shown in pink) to indicate that the unit should not move up, so it continues on its way. Near the top, it detects an obstacle and changes direction. It then finds its way around the "U"-shaped obstacle, following the red path. In contrast, a pathfinder would have scanned a larger area (shown in light blue), but found a shorter path (blue), never sending the unit into the concave shaped obstacle.

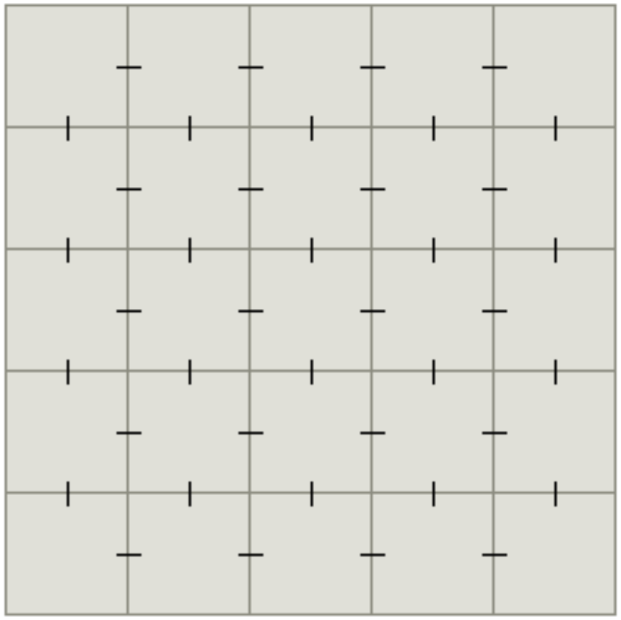
You can however extend a movement algorithm to work around traps like the one shown above. Either avoid creating concave obstacles, or mark their convex hulls as dangerous (to be entered only if the goal is inside):



Pathfinders let you look ahead and make plans rather than waiting until the last moment to discover there's a problem. There's a tradeoff between planning with pathfinders and reacting with movement algorithms. Planning generally is slower but gives better results; movement is generally faster but can get stuck. If the game world is changing often, planning ahead is less valuable. I recommend using both: pathfinding for big picture, slow changing obstacles and long paths; and movement for local area, fast changing, and short paths.

Algorithms

The pathfinding algorithms from computer science textbooks work on *graphs* in the mathematical sense—a set of vertices with edges connecting them. A tiled game map can be considered a graph with each tile being a vertex and edges drawn between tiles that are adjacent to each other:

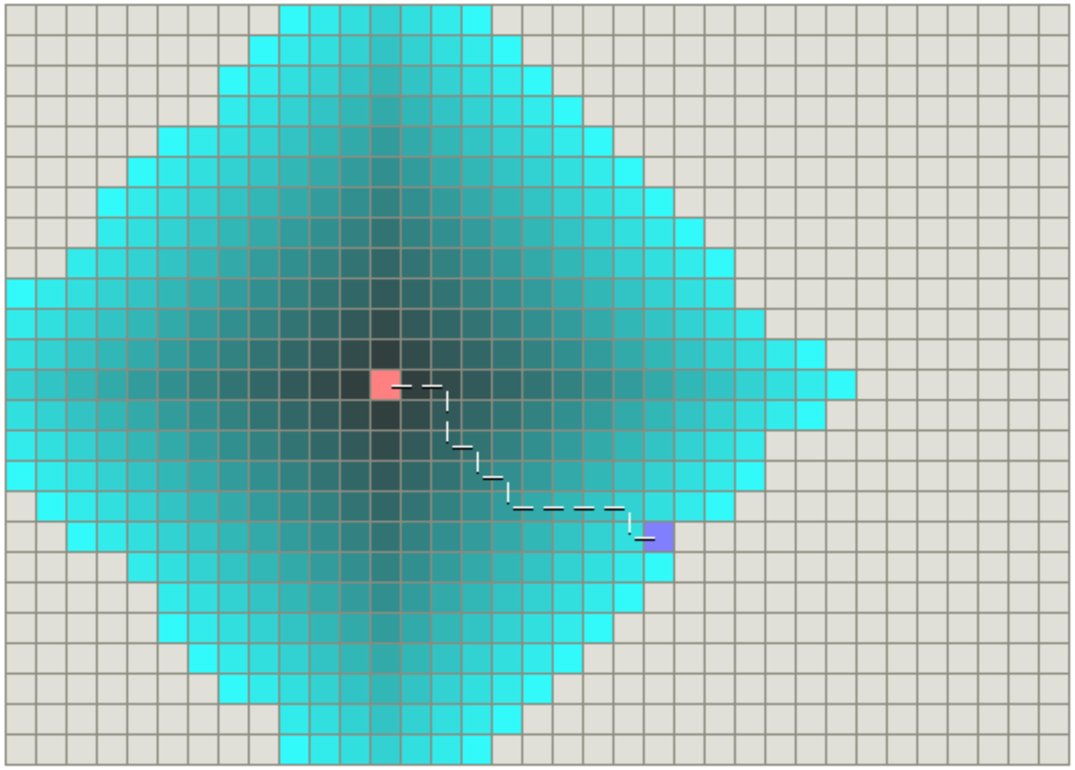


For now, I will assume that we're using two-dimensional grids. Later on, I'll discuss how to build other kinds of graphs out of your game world.

Most pathfinding algorithms from AI or Algorithms research are designed for arbitrary graphs rather than grid-based games. We'd

Dijkstra's Algorithm and Best-First-Search

Dijkstra's algorithm works by visiting vertices in the graph starting with the object's starting point. It then repeatedly examines the closest not-yet-examined vertex, adding its vertices to the set of vertices to be examined. it expands outwards from the starting point until it reaches the goal. Dijkstra's algorithm is guaranteed to find a shortest path from the starting point to the goal, as long as none of the edges have a negative cost. (I write "a shortest path" because there are often multiple equivalently-short paths.) In the following diagram, the pink square is the starting point, the blue square is the goal, and the teal areas show what areas Dijkstra's algorithm scanned. The lightest teal areas are those farthest from the starting point, and thus form the "frontier" of exploration:



Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

```
DIJKSTRA($G, w, s$)
1  INITIALIZE-SINGLE-SOURCE($G, s$)
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN($Q$)
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX($u, v, w$)
```

Dijkstra's algorithm relaxes edges as shown in Figure 24.6. Line 1 initializes the d and π values in the usual way, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the **while** loop of lines 4–8. Line 3 initializes the min-priority queue Q to contain all the vertices in V; since $S = \emptyset$ at that time, the invariant is true after line 3. Each time through the **while** loop of lines 4–8, line 5 extracts a vertex u from $Q = V - S$ and line 6 adds it to set S, thereby maintaining the invariant. (The first time through this loop, $u = s$.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$. Then, lines 7–8 relax each edge (u, v) leaving u, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if we can improve the shortest path to v found so far by going through u. Observe that the algorithm never inserts vertices into Q after line 3 and that each vertex is extracted from Q

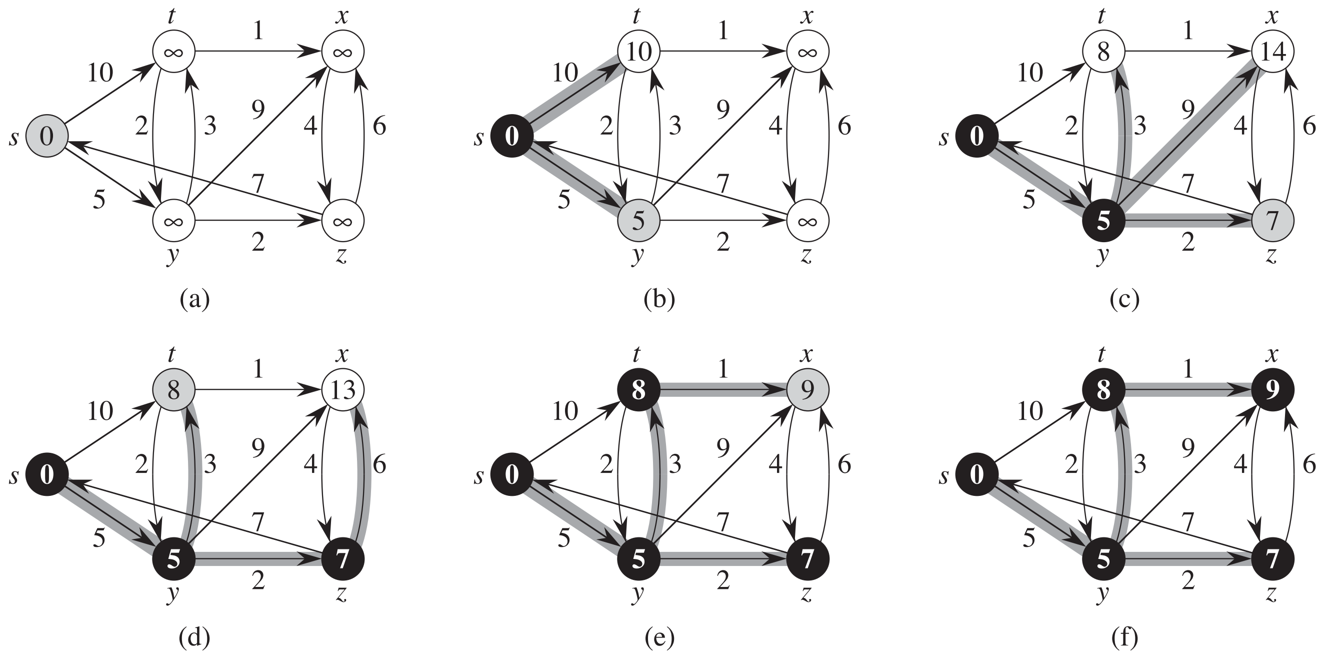


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

and added to S exactly once, so that the **while** loop of lines 4–8 iterates exactly $|V|$ times.

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy. Chapter 16 explains greedy strategies in detail, but you need not have read that chapter to understand Dijkstra's algorithm. Greedy strategies do not always yield optimal results in general, but as the following theorem and its corollary show, Dijkstra's algorithm does indeed compute shortest paths. The key is to show that each time it adds a vertex u to set S, we have $u.d = \delta(s, u)$.

Simple graph representation

[edit]

The data structures supplied by C++'s Standard Template Library facilitate a straightforward implementation of Dijkstra's algorithm. For simplicity, we start with a simple graph representation where the vertices are numbered by sequential integers (0, 1, 2, ...) and the edge weights are `doubles`. We define a `struct` representing an edge that stores its weight and target vertex (the vertex it points to):

```
<<simple graph types>>=
typedef int vertex_t;
typedef double weight_t;

struct edge {
    const vertex_t target;
    const weight_t weight;
    edge(vertex_t arg_target, weight_t arg_weight)
        : target(arg_target), weight(arg_weight) { }
};
```

Because we'll need to iterate over the successors of each vertex, we will find an adjacency list representation most convenient. We will represent this as an *adjacency map*, a mapping from each vertex to the list of edges exiting that vertex, as defined by the following `typedef`:

```
<<simple graph types>>=
typedef std::map<vertex_t, std::list<edge> > adjacency_map_t;

<<definition headers>>=
#include <map>
#include <list>
```

```
<<dijkstra_example.cpp>>=
#include <iostream>
#include <vector>
#include <string>
definition headers

simple graph types

simple compute paths function

get shortest path function

int main()
{
    adjacency_map_t adjacency_map;
    std::vector<std::string> vertex_names;

    initialize adjacency map

    std::map<vertex_t, weight_t> min_distance;
    std::map<vertex_t, vertex_t> previous;
    DijkstraComputePaths(0, adjacency_map, min_distance, previous);
    print out shortest paths and distances
    return 0;
}
```

Printing out shortest paths is just a matter of iterating over the vertices and calling `DijkstraGetShortestPathTo()` on each:

```
<<simple compute paths function>>=
void DijkstraComputePaths(vertex_t source,
                          const adjacency_map_t &adjacency_map,
                          std::map<vertex_t, weight_t> &min_distance,
                          std::map<vertex_t, vertex_t> &previous)
{
    initialize output parameters
    min_distance[source] = 0;
    visit each vertex u, always visiting vertex with smallest min_distance first
        // Visit each edge exiting u
        const std::list<edge> &edges = adjacency_map.find(u)->second;
        for (std::list<edge>::const_iterator edge_iter = edges.begin();
             edge_iter != edges.end();
             edge_iter++)
        {
            vertex_t v = edge_iter->target;
            weight_t weight = edge_iter->weight;
            relax the edge (u,v)
        }
    }
}
```

This completes `DijkstraComputePaths()`. `DijkstraGetShortestPathTo()` is much simpler, just following the linked list in the `previous` map from the target back to the source:

```
<<get shortest path function>>=
std::list<vertex_t> DijkstraGetShortestPathTo(
    vertex_t target, const std::map<vertex_t, vertex_t> &previous)
{
    std::list<vertex_t> path;
    std::map<vertex_t, vertex_t>::const_iterator prev;
    vertex_t vertex = target;
    path.push_front(vertex);
    while((prev = previous.find(vertex)) != previous.end())
    {
        vertex = prev->second;
        path.push_front(vertex);
    }
    return path;
}
```

Printing out shortest paths is just a matter of iterating over the vertices and calling `DijkstraGetShortestPathTo()` on each:

```
<<print out shortest paths and distances>>=
for (adjacency_map_t::const_iterator vertex_iter = adjacency_map.begin();
     vertex_iter != adjacency_map.end();
     vertex_iter++)
{
    vertex_t v = vertex_iter->first;
    std::cout << "Distance to " << vertex_names[v] << ": " << min_distance[v] << std::endl;
    std::list<vertex_t> path =
        DijkstraGetShortestPathTo(v, previous);
    std::list<vertex_t>::const_iterator path_iter = path.begin();
    std::cout << "Path: ";
    for( ; path_iter != path.end(); path_iter++)
    {
        std::cout << vertex_names[*path_iter] << " ";
    }
    std::cout << std::endl;
}
```

For this example, we choose vertices corresponding to some East Coast U.S. cities. We add edges corresponding to interstate highways, with the edge weight set to the driving distance between the cities in miles as determined by Mapquest (note that edges are directed, so if we want an "undirected" graph, we would need to add edges going both ways):

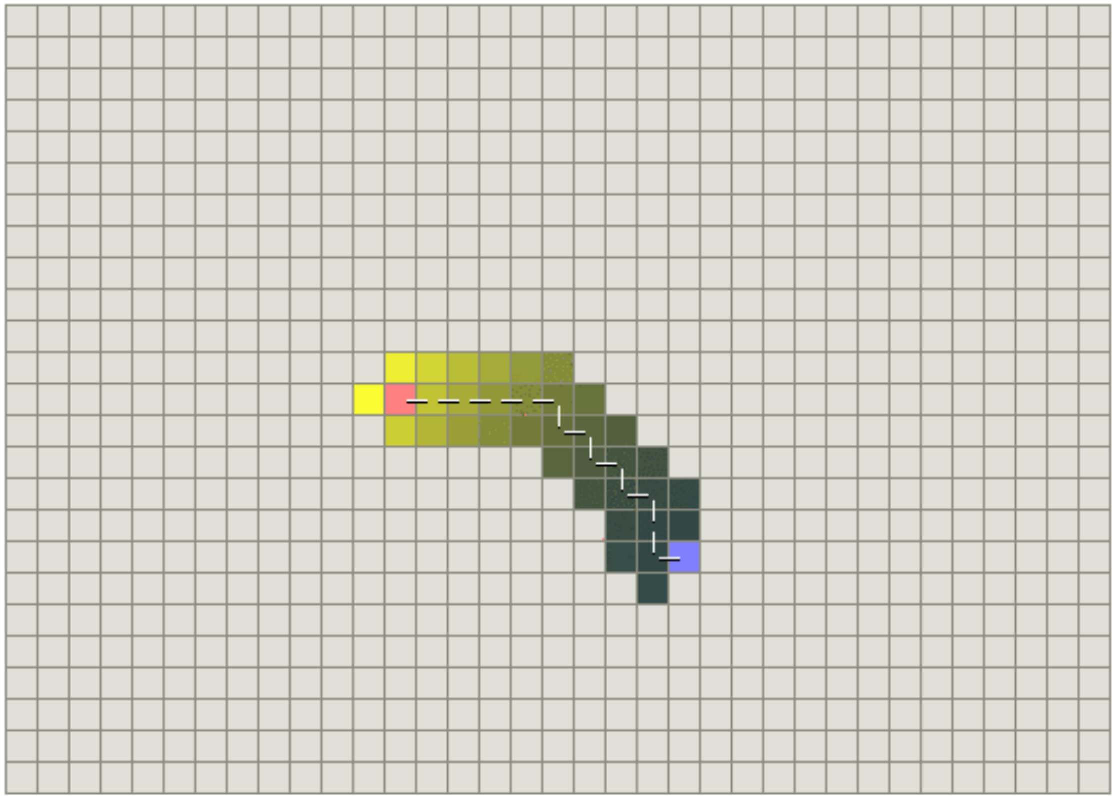
```
<<initialize adjacency map>>=
vertex_names.push_back("Harrisburg");   // 0
vertex_names.push_back("Baltimore");    // 1
vertex_names.push_back("Washington");   // 2
vertex_names.push_back("Philadelphia"); // 3
vertex_names.push_back("Binghamton");   // 4
vertex_names.push_back("Allentown");    // 5
vertex_names.push_back("New York");     // 6
adjacency_map[0].push_back(edge(1,  79.83));
adjacency_map[0].push_back(edge(5,  81.15));
adjacency_map[1].push_back(edge(0,  79.75));
```

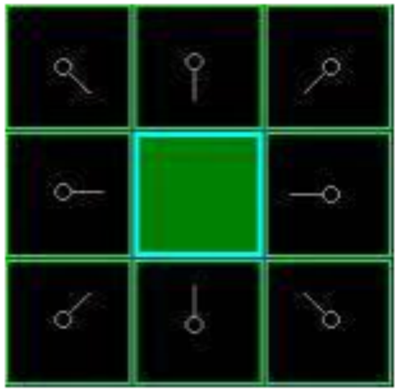
```
/* Retrieved from:
http://en.literateprograms.org/Dijkstra's_algorithm_(C_Plus_Plus)?oldid=17819 */

#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <list>
#include <limits> // for numeric_limits
#include <set>
#include <utility> // for pair

typedef int vertex_t;
typedef double weight_t;

struct edge {
   const vertex_t target;
   const weight_t weight;
   edge(vertex_t arg_target, weight_t arg_weight)
      : target(arg_target), weight(arg_weight) { }
};

typedef std::map<vertex_t, std::list<edge> > adjacency_map_t;


void DijkstraComputePaths(vertex_t source,
                          const adjacency_map_t &adjacency_map,
                          std::map<vertex_t, weight_t> &min_distance,
                          std::map<vertex_t, vertex_t> &previous)
{
   for (adjacency_map_t::const_iterator vertex_iter = adjacency_map.begin();
        vertex_iter != adjacency_map.end();
        vertex_iter++)
   {
      vertex_t v = vertex_iter->first;
      min_distance[v] = std::numeric_limits< double >::infinity();
      for (std::list<edge>::const_iterator edge_iter = vertex_iter->second.begin();
           edge_iter != vertex_iter->second.end();
           edge_iter++)
      {
         vertex_t v2 = edge_iter->target;
         min_distance[v2] = std::numeric_limits< double >::infinity();
      }
   }
   min_distance[source] = 0;
```

OUTPUT:

```
g++ dijkstra_example.cpp
dhcp-10-7-2-227:TurbineWarnerBros russell$ ./a.out
Distance to Harrisburg: 0
Path: Harrisburg
Distance to Baltimore: 79.83
Path: Harrisburg Baltimore
Distance to Washington: 119.25
Path: Harrisburg Baltimore Washington
Distance to Philadelphia: 143.2
Path: Harrisburg Allentown Philadelphia
Distance to Binghamton: 215.62
Path: Harrisburg Allentown Binghamton
Distance to Allentown: 81.15
Path: Harrisburg Allentown
Distance to New York: 172.78
Path: Harrisburg Allentown New York
```

The Greedy Best-First-Search algorithm works in a similar way, except that it has some estimate (called a *heuristic*) of how far from the goal any vertex is. Instead of selecting the vertex closest to the starting point, it selects the vertex closest to the goal. (Greedy) Best-First-Search is *not* guaranteed to find a shortest path. However, it runs much quicker than Dijkstra's algorithm because it uses the heuristic function to guide its way towards the goal very quickly. For example, if the goal is to the south of the starting position, Best-First-Search will tend to focus on paths that lead southwards. In the following diagram, yellow represents those nodes with a high heuristic value (high cost to get to the goal) and black represents nodes with a low heuristic value (low cost to get to the goal). It shows that Best-First-Search can find paths very quickly compared to Dijkstra's algoritm:




[Figure 2]

http://aitopics.org/topic/games-puzzles

Next, we choose one of the adjacent squares on the open list and more or less repeat the earlier process, as described below. But which square do we choose? The one with the lowest F cost.

Path Scoring

The key to determining which squares to use when figuring out the path is the following equation:

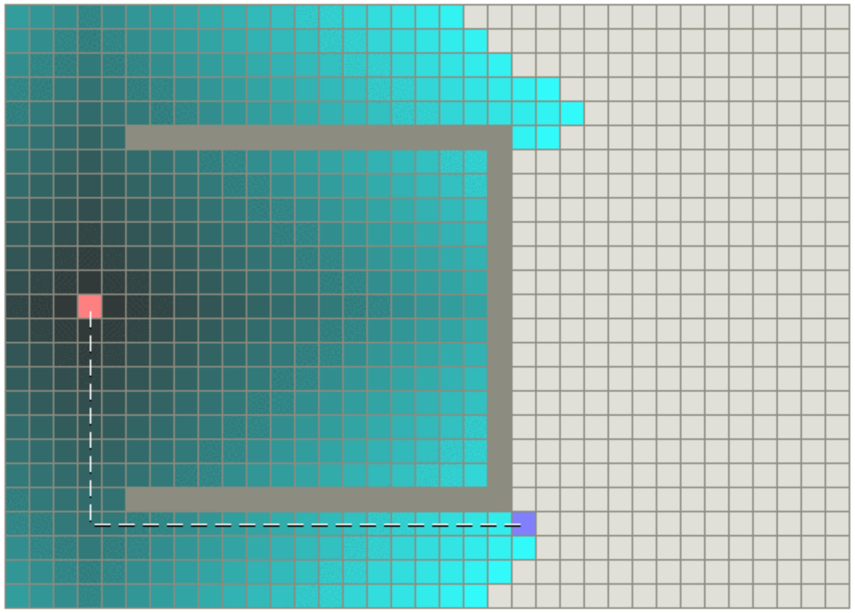
F = G + H

where

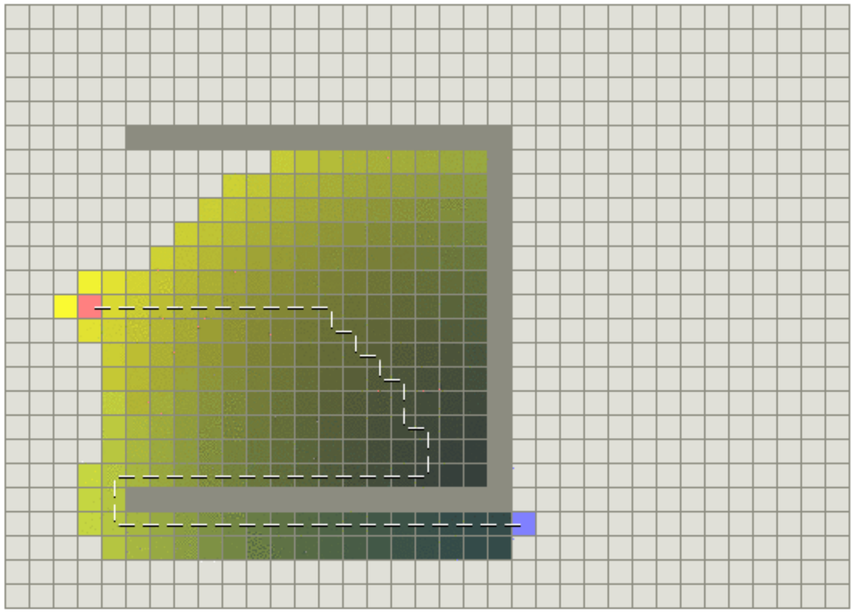
- G = the movement cost to move from the starting point A to a given square on the grid, following the path generated to get there.
- H = the estimated movement cost to move from that given square on the grid to the final destination, point B. This is often referred to as the heuristic, which can be a bit confusing. The reason why it is called that is because it is a guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). You are given one way to calculate H in this tutorial, but there are many others that you can find in other articles on the web.

Our path is generated by repeatedly going through our open list and choosing the square with the lowest F score. This process will be described in more detail a bit further in the article. First let's look more closely at how we calculate the equation.

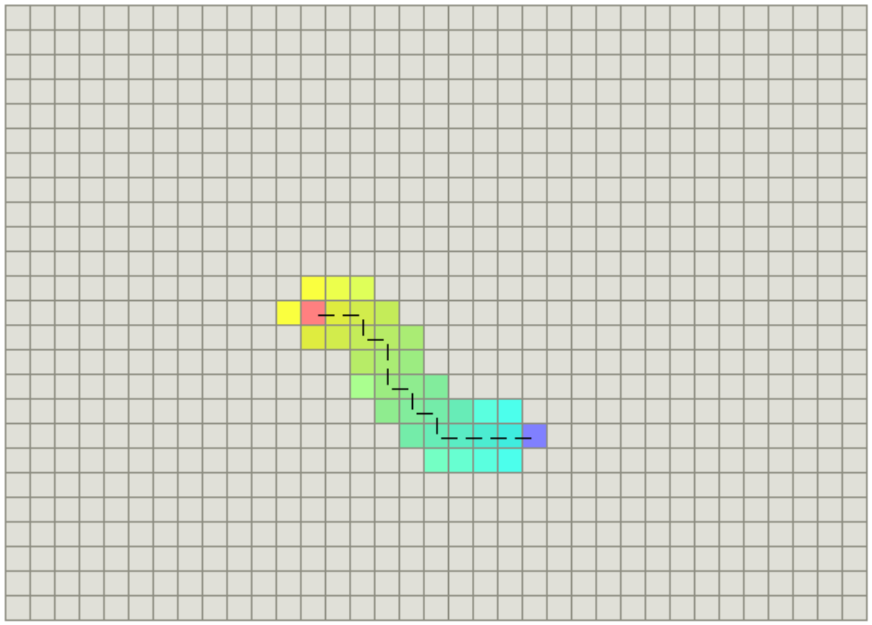
described in the previous section. Dijkstra's algorithm works harder but is guaranteed to find a shortest path:



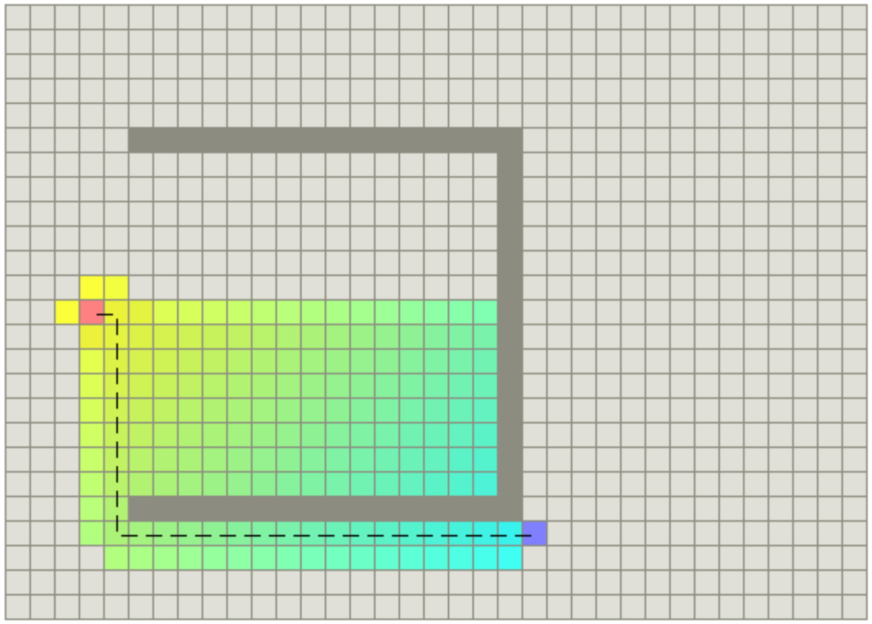
Best-First-Search on the other hand does less work but its path is clearly not as good:



itself. In the simple case, it is as fast as Best-First-Search:



In the example with a concave obstacle, A* finds a path as good as what Dijkstra's algorithm found:



The heuristic function `h(n)` tells A* an *estimate* of the minimum cost from any vertex `n` to the goal. It's important to choose a good heuristic function.

A*'s Use of the Heuristic

The heuristic can be used to control A*'s behavior.

- At one extreme, if `h(n)` is 0, then only `g(n)` plays a role, and A* turns into Dijkstra's algorithm, which is guaranteed to find a shortest path.
- If `h(n)` is always lower than (or equal to) the cost of moving from `n` to the goal, then A* is guaranteed to find a shortest path. The lower `h(n)` is, the more node A* expands, making it slower.
- If `h(n)` is exactly equal to the cost of moving from `n` to the goal, then A* will only follow the best path and never expand anything else, making it very fast. Although you can't make this happen in all cases, you can make it exact in some special cases. It's nice to know that given perfect information, A* will behave perfectly.
- If `h(n)` is sometimes greater than the cost of moving from `n` to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.
- At the other extreme, if `h(n)` is very high relative to `g(n)`, then only `h(n)` plays a role, and A* turns into Best-First-Search.

So we have an interesting situation in that we can decide what we want to get out of A*. At exactly the right point, we'll get shortest paths really quickly. If we're too low, then we'll continue to get shortest paths, but it'll slow down. If we're too high, then we give up shortest paths, but A* will run faster.

```
public void findBestPath() {
        System.out.println("Calculating best path...");
        Set<Square> adjacencies = elements[0][0].getAdjacencies();
        for (Square adjacency : adjacencies) {
                adjacency.setParent(elements[0][0]);
                if (adjacency.isStart() == false) {
                        opened.add(adjacency);
                }
        }
        while (opened.size() > 0) {
                Square best = findBestPassThrough();
                opened.remove(best);
                closed.add(best);
                if (best.isEnd()) {
                        System.out.println("Found Goal");
                        populateBestList(goal);
                        draw();
                        return;
                } else {
                        Set<Square> neighbors = best.getAdjacencies();
                        for (Square neighbor : neighbors) {
                                if (opened.contains(neighbor)) {
                                        Square tmpSquare = new Square(neighbor.getX(),
                                                        neighbor.getY(), this);
                                        tmpSquare.setParent(best);
                                        if (tmpSquare.getPassThrough(goal) >= neighbor
                                                        .getPassThrough(goal)) {
                                                continue;
                                        }}
                                if (closed.contains(neighbor)) {
                                        Square tmpSquare = new Square(neighbor.getX(),
                                                        neighbor.getY(), this);
                                        tmpSquare.setParent(best);
                                        if (tmpSquare.getPassThrough(goal) >= neighbor
                                                        .getPassThrough(goal)) {
                                                continue;
                                        }
                                }
                                neighbor.setParent(best);
                                opened.remove(neighbor);
                                closed.remove(neighbor);
                                opened.add(0, neighbor);
                        } } }
        System.out.println("No Path to goal");
}
```

Tips to avoid common game AI mistakes

“9) Using A* for everything

I run a web site which teaches A*, and yet when I come to write a pathfinding or other search program I don't always use A*, and when I do I never do A* on a fine mesh or grid. Why? Because it uses a lot of memory and processing power that is totally unneccesary.

Firstly if you're searching only small data sets, like an AI state graph, or a dozen or so path nodes in a room, you can use Dijkstra's algorithm, which is particularly effective when all the AI's in the room will path find to the same node, since the algorithm will fill out data for the whole network rather than just the start and end nodes. Sometimes even a breadthfirst search is enough.

In general you want the path finding data to be as high level as possible whilst still allowing movement to all possible gameplay areas.

One approach is hierarchical path finding, which really needs to work at engine level. If you divide your game world up into regions, buildings, floors and rooms, for example, an AI can path find at all those levels before finally pathfinding on a grid or mesh level inside the current room it is in.”

QuickSort O(n ln(n))

Quicksort, like merge sort, applies the divide-and-conquer paradigm introduced in Section 2.3.1. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p \mathinner{..} r]$:

Divide: Partition (rearrange) the array $A[p \mathinner{..} r]$ into two (possibly empty) subarrays $A[p \mathinner{..} q-1]$ and $A[q+1 \mathinner{..} r]$ such that each element of $A[p \mathinner{..} q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1 \mathinner{..} r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p \mathinner{..} q-1]$ and $A[q+1 \mathinner{..} r]$ by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p \mathinner{..} r]$ is now sorted.

The following procedure implements quicksort:

```
QUICKSORT($A, p, r$)
1  if $p < r$
2      $q = $ PARTITION($A, p, r$)
3      QUICKSORT($A, p, q - 1$)
4      QUICKSORT($A, q + 1, r$)
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, $A.length$).

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \,..\, r]$ in place.

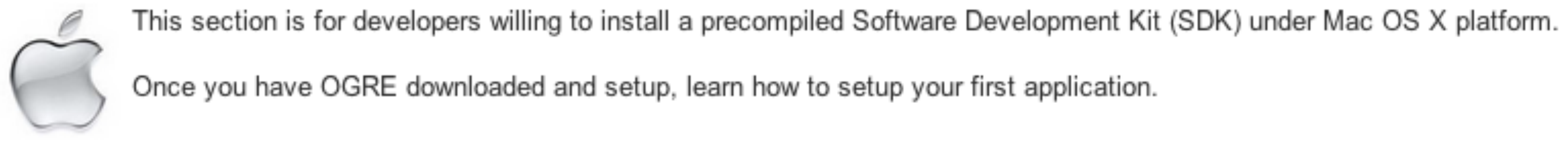
```
PARTITION($A, p, r$)
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4      if $A[j] \leq x$
5          $i = i + 1$
6          exchange $A[i]$ with $A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

How to install boost & Ogre

Mac:
Run the following command:

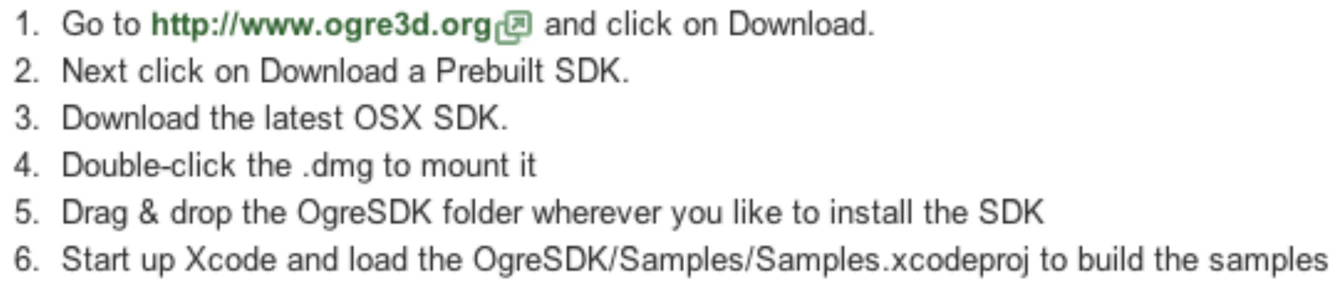
```
sudo port install boost
```

Windows:
http://www.boost.org/users/download/



This section is for developers willing to install a precompiled Software Development Kit (SDK) under Mac OS X platform.

Once you have OGRE downloaded and setup, learn how to setup your first application.



1. Go to **http://www.ogre3d.org** and click on Download.
2. Next click on Download a Prebuilt SDK.
3. Download the latest OSX SDK.
4. Double-click the .dmg to mount it
5. Drag & drop the OgreSDK folder wherever you like to install the SDK
6. Start up Xcode and load the OgreSDK/Samples/Samples.xcodeproj to build the samples

```
//=====================================================================
#include <boost/config.hpp>
#include <iostream>
#include <fstream>

#include <boost/graph/graph_traits.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>

using namespace boost;

Int main(int, char *[])
{
  typedef adjacency_list < listS, vecS, directedS,
    no_property, property < edge_weight_t, int > > graph_t;
  typedef graph_traits < graph_t >::vertex_descriptor vertex_descriptor;
  typedef graph_traits < graph_t >::edge_descriptor edge_descriptor;
  typedef std::pair<int, int> Edge;

  const int num_nodes = 5;
  enum nodes { A, B, C, D, E };
  char name[] = "ABCDE";
  Edge edge_array[] = { Edge(A, C), Edge(B, B), Edge(B, D), Edge(B, E),
    Edge(C, B), Edge(C, D), Edge(D, E), Edge(E, A), Edge(E, B)
  };
  int weights[] = { 1, 2, 1, 2, 7, 3, 1, 1, 1 };
  int num_arcs = sizeof(edge_array) / sizeof(Edge);
#if defined(BOOST_MSVC) && BOOST_MSVC <= 1300
  graph_t g(num_nodes);
  property_map<graph_t, edge_weight_t>::type weightmap = get(edge_weight, g);
  for (std::size_t j = 0; j < num_arcs; ++j) {
    edge_descriptor e; bool inserted;
    tie(e, inserted) = add_edge(edge_array[j].first, edge_array[j].second, g);
    weightmap[e] = weights[j];
  }
#else
  graph_t g(edge_array, edge_array + num_arcs, weights, num_nodes);
  property_map<graph_t, edge_weight_t>::type weightmap = get(edge_weight, g);
#endif
  std::vector<vertex_descriptor> p(num_vertices(g));
  std::vector<int> d(num_vertices(g));
  vertex_descriptor s = vertex(A, g);

#if defined(BOOST_MSVC) && BOOST_MSVC <= 1300
  // VC++ has trouble with the named parameters mechanism
#if defined(BOOST_MSVC) && BOOST_MSVC <= 1300
  // VC++ has trouble with the named parameters mechanism
  property_map<graph_t, vertex_index_t>::type indexmap = get(vertex_index, g);
  dijkstra_shortest_paths(g, s, &p[0], &d[0], weightmap, indexmap,
                          std::less<int>(), closed_plus<int>(),
                          (std::numeric_limits<int>::max)(), 0,
                          default_dijkstra_visitor());
#else
  dijkstra_shortest_paths(g, s, predecessor_map(&p[0]).distance_map(&d[0]));
#endif

  std::cout << "distances and parents:" << std::endl;
  graph_traits < graph_t >::vertex_iterator vi, vend;
  for (tie(vi, vend) = vertices(g); vi != vend; ++vi) {
    std::cout << "distance(" << name[*vi] << ") = " << d[*vi] << ", ";
    std::cout << "parent(" << name[*vi] << ") = " << name[p[*vi]] << std::
      endl;
  }
  std::cout << std::endl;

  std::ofstream dot_file("figs/dijkstra-eg.dot");

  dot_file << "digraph D {\n"
    << "  rankdir=LR\n"
    << "  size=\"4,3\"\n"
    << "  ratio=\"fill\"\n"
    << "  edge[style=\"bold\"]\n" << "  node[shape=\"circle\"]\n";

  graph_traits < graph_t >::edge_iterator ei, ei_end;
  for (tie(ei, ei_end) = edges(g); ei != ei_end; ++ei) {
    graph_traits < graph_t >::edge_descriptor e = *ei;
    graph_traits < graph_t >::vertex_descriptor
      u = source(e, g), v = target(e, g);
    dot_file << name[u] << " -> " << name[v]
      << "[label=\"" << get(weightmap, e) << "\"";
    if (p[v] == u)
      dot_file << ", color=\"black\"";
    else
      dot_file << ", color=\"grey\"";
    dot_file << "]";
  }
  dot_file << "}";
  return EXIT_SUCCESS;
}
```

Simple Tic-Tac-Toe “AI”

Basic Tic-Tac-Toe strategy for the computer:

1) If there's a move that allows the computer to win this turn, the computer should choose that move.
2) If there's a move that allows the human to win next turn, the computer should choose that move.
3) Otherwise, the computer should choose the best empty square as its move. The best square is the center. The next best squares are the corners. And the next best squares are the corners. And the next best squares are the rest.

```python
def instructions():
    """ Display game instructions."""
    print \
    """
    Welcome to the greatest intellectual challenge of all time: Tic-Tac-Toe.
    This will be a showdown between your human brain and my silicon processor.

    You will make your move known by entering a number, 0 - 8. The number
    will correspond to the board position as illustrated:

                     0 | 1 | 2
                    -----------
                     3 | 4 | 5
                    -----------
                     6 | 7 | 8

    Prepare yourself, human. The ultimate battle is about to begin. \n
    """
    # if computer can win, take that move
    for move in legal_moves(board):
        board[move] = computer
        if winner(board) == computer:
            print move
            return move
        # done checking this move, undo it
        board[move] = EMPTY
```

If I get to this point in the function, it means the computer can't win on its next move. So, I check to see if the player can win on his or her next move. The code loops through the list of the legal moves, putting the human's piece in each empty square, checking for a win. If the human can win, then that's the move to take for a block. If this is the case, the function returns the move and ends. Otherwise, I undo the move and try the next legal move in the list.

```
    # if human can win, block that move
    for move in legal_moves(board):
        board[move] = human
        if winner(board) == human:
            print move
            return move
        # done checking this move, undo it
        board[move] = EMPTY
```

If I get to this point in the function, then neither side can win on its next move. So, I look through the list of best moves and take the first legal one. The computer loops through `BEST_MOVES`, and as soon as it finds one that's legal, it returns that move.

```
    # since no one can win on next move, pick best open square
    for move in BEST_MOVES:
        if move in legal_moves(board):
            print move
            return move
```

ENDE