Robotic Motion Planning Applied to Word Ladder Problems

by

Brent M. Dingle

Spring 2002

Abstract:
This paper discusses a simple way of formulating a word puzzle in terms that can be solved using a Motion Planner. The purpose is to illustrate the potential use of already existing planning techniques towards new sets of problems by creatively and correctly restating problems in terms that apply. The intent is to demonstrate to beginning Computer Scientists a method of approaching problems by applying already known solutions, rather than reinventing solutions. This paper does not claim to be solving this problem in any "better" way, just a different way. The intended audience is introductory level Computer Science students already familiar with motion planning techniques.

INTRODUCTION

We have all seen word ladder problems. For example if the puzzle was: transform FOOL to WISE altering one letter at each step with a restriction that at each step you must be at a valid word you might use the sequence:

```
FOOL
POOL
POLL
PILL
WILL
WILE
WISE
```

Most of the time the problems also limit the number of steps you are to use. In case you wish to brush up on your skills attempt the following. In 5 steps transform BUSH to GORE, altering only one letter at each step with the restriction that each step must be a valid word.

```
G  O  R  E
__ __ __ __
__ __ __ __
__ __ __ __
__ __ __ __

B  U  S  H
```

So how does this relate to Motion Planning?

THE PROBLEM

In Motion Planning there is always a configuration space and the goal is to move from a start configuration to an end configuration, usually in some "shortest distance." This should seem very similar to the above, but what would our configuration space be?

Well, if we remove the "each step must be a word" condition we quickly discover any problem can be solved in *n* steps, where *n* would be the length of the two words. So we might approach the problem from that angle and allow the configuration space to be all permutations of *n* letters. However that makes the problem trivial. So we need to keep the "each step must be a word" condition.

This should immediately cause one to notice that not all puzzles can be solved in *n* steps. This leads to the conclusion that our actual configuration space is all *n*-letter words. Notice this does not make the solutions trivial as we must also meet the restriction of only changing one letter from step to step. Thus we discover a distance measurement idea, which may also lead to an idea of obstacles.

So let us consider two words, W1 and W2, both of length *n*, where W1[i] denotes the ith letter of W1. Let us define the distance between the two words as the sum of the letters such that W1[i] ≠ W2[i] for all i = 1 to *n*. Thus the distance between the words PILL and POLL is one, whereas the distance from PILL to WISE is three. Notice this should be understood to be the minimum distance. The actual distance may be much larger, for the actual distance will be defined as the (minimum) number of steps to actually transform a word into another word under the constraints given above.

With these definitions of distance we see we could solve any problem by making a table of $O(n^2)$ values based on a dictionary of *n*-length words. And given any word simply chart all paths from the word following trails of words such that from the table the distance from any two words along a path is exactly one.

If our dictionary of words is small then the above is a small problem and we will likely not be able to solve many problems as we will not have enough words to do so. However, if our dictionary is large then we are faced with a more difficult problem.

SETUP FOR A PRM

To summarize, for word ladder problems, we define our configuration space, C, to be all permutations of *n* letters. Then, our free configuration space, C_{free}, is all words in our dictionary of length *n*. Our obstacle configuration space, C_{obs}, is all *n* letter permutations not in our dictionary. Our movement constraint is that we can only move from one word to another if the distance between them is one (i.e. the words are only different in exactly one letter).

We will now present a way to solve these word ladder problems using a motion planning technique referred to as the Probabilistic Roadmap Method (PRM). It should be noted that this problem can be solved in other ways and using different motion planning methods. However, our goal is not to prove that this is the best method nor the only solution. We mean only to show that PRMs can be used to solve such a problem. We want to emphasize the issue of redefining the problem in such a way as to be able to apply this method. We believe this type of problem solving technique is crucial to advancing Computer Science.

In this method we construct a representation of C_{free} using an undirected graph G(*V*, *E*), where *V* is a set of vertices and *E* is a set of edges. Each vertex in *V* is a member of C_{free}. In our case each member of *V* is a word in our dictionary of length *n*. Each edge in *E* is a collision free path from a pair of vertices in *V*. Notice also that each of our edges will only connect vertices which are exactly a distance of one apart from one another.

In most PRM methods the graph G is preconstructed before any queries are run (i.e. we construct G before trying to solve any puzzle). Once G is constructed we should be able to use it to solve any puzzle (corresponding to the word length = *n*).

The general algorithm for a Probabilistic Roadmap Method is as follows [Lavalle]:

```
1  G.initialize();
2  **for** $i$=1 **to** $M$
3      q ← RandFreeConfig($q$);
4      G.addVertex($q$);
5      **for each** v ∈ NhbrHd($q$,G)
6          **if** ((**not** G.sameComponent($q$,$v$)) **and** Connect($q$,$v$)) **then**
7              G.addEdge($q$,$v$);
8  Return G;
```

The returned graph will have *M* vertices. The functions of the above algorithm are described below. It might be noted that the condition (**not** G.sameComponent(*q*,*v*)) is sometimes replaced with the condition (G.vertexDegree(*q*) < *K*), for some fixed *K*.

The RandFreeConfig() function is used to find a random configuration in C_{free}. In our case this is trivial as all we need to do is select a random word (of length *n*) from our dictionary. For illustrative purposes the general construct of the algorithm is:

```
1  **repeat**
2      q ← GetRandConfig();
3  **until** q ∈ $C_{free}$;
4  Return q;
```

The NhbrHd() function returns an ordered list of vertices in G(*V*, *E*). Each of these vertices (words) are within a distance ε of *q*. Where ε, and the distance metric ρ(*q*, *v*) are predetermined. In our case ρ(*q*, *v*) is the number of letters the word corresponding to vertex *q* is different from the word corresponding to *v*. And if we set ε equal to one, the Connect(q, v) becomes automatic.

The G.sameComponent() condition will determine if *q* and *v* are in the same connected component of *G*. If they are in the same component then we usually do not try to connect them as there is already a path from one to the other. However, if we are seeking shortest paths it may be required. (In what follows, we will not be seeking shortest paths).

The general algorithm for determining if *q* and *v* are in the same connected component is defined below. It is assumed that Q is a priority queue with its elements in increasing order of distance from vertex *q*.

```
1 Q.init();
2 for each v ∈ V
3     if ρ(q, v) < ε then
4         Q.insert(v);
5 Return q;
```

The Connect(*q*,*v*) function is sometimes referred to as a local planner. Basically this function attempts to connect *q* and *v* using a very simple and quick algorithm. If it is unable to connect them it returns false. If it succeeds in connecting them it returns true. Notice this does not necessarily mean that *q* and *v* are adjacent in the graph. It just means there is a path from one to the other. According to the general algorithm this means we must place an edge between *q* and *v*. For our purposes however it will be better to add the connecting path to G. As an aside, notice another option might be to weight the edges and any weight greater than one would require a call to the local planner in the query phase.

So our Connect function will take several parameters including *q, v, G*. It will return true if it found a path (a mini-ladder) from *q* to *v* and it will insert the appropriate connections and vertices into G. If no such path is found it will return false. We may want to further limit it to not trying to go more than a certain distance ($n/2 + 1$). If it goes farther than that it will be solving a problem harder than our general problem. This may lead to some solutions being missed and may require tweaking. Finding good local planners is often not easy.

CREATING THE MAIN MAP

So to set up our main map we would use the following pseudocode:

```
PROCEDURE PRM:
      INPUT:
            Graph Pointer (the map),
            Input Dictionary filename,
            The desired number of vertices to be placed in the map,
            The maximum degree of each vertex
      OUTPUT:
            Graph Pointer will be the main PRM map
{
   Initialize Graph to have zero vertices/nodes;
   while (i <= num_verts)
   {
      node = GetRandomNode(...);  // from input dictionary file
      graph->AddNode(node);
      increment i by one;
      CheckNhbrs(...) // For each node v (valid word), whose distance
            is one from node add edge from v to node - also add v into
            graph if not already in graph and increment i so our vertex
            count is correct.
   }
} // end PRM
```

The procedure CheckNhbrs is a slight variation from the standard concept of PRM methods. What it basically does is “bush out” our randomly selected nodes. For example if our randomly selected node was cool, then CheckNhbrs would likely add pool, fool, tool and wool to our PRM main map. Below is the pseudocode for it:

```
PROCEDURE CheckNhbrs
      INPUT:
            Dictionary Input File
            Word length
            Graph pointer (the map being created)
            Node Pointer
            Maximum/Desired Number of Vertices to be placed in the map
      OUTPUT:
            Any words one distant from the node’s word will be added to
            Graph
{
   // loop through each character of node’s word
   for (let_index = 0; let_index < num_lets; let_index++)
   {
      Change the letter to a, b, c, ... z
      If this creates a word found in the dictionary add it to Graph
      and check if it is also a nhbr of anything else already in Graph
   }

} // end CheckNhbrs
```

So the above pseudocode would establish our PRM map. From this map we should be able to create a word ladder from any given word to any other given word (assuming our input dictionary allows it). Notice this is only a **should be able to** statement. If we put all the words in our dictionary into the initial map it would become a guaranteed statement. However, one objective of PRM methods is not to require all free configuration states to be included in the main map, hence it remains a should be able to statement.

LOCAL PLANNING

Since not all of our words (not all of our free configuration states) are included in the main PRM map it is likely we might choose an initial word not in it. When this occurs we must use a local planning method to create a path from our chosen word to the PRM map.

The below pseudocode illustrates a trivial local planner which will add the word to our map. But it will only be connected if it happens to be a distance of one from any of the words already in the map.

```
FUNCTION Connect
      INPUT:
            node addme
      OUTPUT:
      Graph with node added and connected the pre-existing map.
{
   Add the node to the graph

   Cycle through all members already in list
      If any are within 1 to dist characters of addme
      Then Make them Neighbors

} // end Connect
```

For example say our PRM main map was something like the following, notice it has only one connected component:

```
cool – pool – fool – tool – wool
              /     \
          foul       fowl
```

Also assume the words bowl and bats are in our dictionary, but not selected to be in our PRM main map.

Now suppose our initial words were cool and bowl. Nothing would need to be done for cool, as it already is in the map. However, we would need to insert bowl into the map and using the above described local planner attempt to connect it to the component(s) of the already existing map.

So our code would examine each word already in the map. If any word was found to be only one letter different from bowl, then it and bowl would be connected. Thus our code would connect bowl to fowl. Therefore if we asked for a word ladder from cool to bowl our code would likely return: cool, pool, fool, fowl, bowl.

Notice our code does not necessarily return the most optimal word ladder. However this could be fixed with a bit of optimizing code or better graph construction techniques.

Another quickly realized problem is that our local planner might be too trivial. For example if our words were cool and bows we would be unable to find a path. Yet if our local planner connected words a distance of TWO together instead of just one, we would succeed – since we can go from bows to bowl.

Problems such as these are difficulties to be dealt with constructing good PRMs and local planners. Regardless using the above local planner with the already described PRM code we could solve word ladder problems using PRM techniques.

CONCLUSION

So we have now seen that the word ladder problem may be restated in such a way as to allow for motion planning techniques to be applied to it. While it is not the most efficient way, nor does it necessarily guarantee a solution, it does work. This would leave room to investigate why it does not always work and how that might be achieved. For example it is interesting to play with how many words need to be included in the main PRM map. When a solution actually existed for four letter words it seemed in our trials that ¼ of our roughly 6000 word dictionary was sufficient to achieve success. It would also be interesting to develop optimizing routines to shorten the ladder as much as possible. But we may also want to go beyond this to applying what we have learned to other problems.

The above is just one way to solve a very specific problem using a motion planning technique. This may lead us to consider other problems that may also be restated in such a way as to apply motion planning techniques. In particular we may apply the same technique described above to other problems involving easily identified 'valid' states where only a finite number of changes can occur between states. For example consider the transition states of a computer (a program, an operating system, a compiler…) or database queries. Perhaps these problems will be addressed in future papers.

As a final wrap up, in case the practice problem in the introduction has you frustrated here is the solution: BUSH, PUSH, POSH, POSE, PORE, GORE. You will also find the full source code and dictionaries for our experiments on which this paper is based at: http://people.cs.tamu.edu/dingle/Projects/

BIBLIOGRAPHY

Amato1998	N. M. Amato, O. B. Bayazit, L. K. Dale, C. V. Jones, and D. Vallejo. *Choosing good distance metrics and local planners for probabilistic roadmap methods*. Technical Report 98-010, Dept. of Computer Science, Texas A&M University, May 1998. A preliminary version of this paper appeared in ICRA'98.
Bohlin2000	R. Bohlin and L. E. Kavraki. *Path planning using lazy PRM*. In Proc. IEEE Int. Conf. Rob. & Autom., pages 521--528, 2000.
Branicky2001	M. Branicky, S. Lavalle, K. Olson and L. Yang. *Quasirandomized path planning*. IEEE International Conference on Robotics and Automation, Seoul(Korea), 2001.
Glavine1990	B. Glavina. *Solving findpath by combination of directed and randomized search*. In Proc. IEEE Internat. Conf. Robot. Autom., pages 1718 – 1723, 1990.
Kavraki1994	L. Kavraki and J.-C. Latombe. *Randomized preprocessing of configuration space for fast path planning*. In Proc. IEEE Int. Conf. Robotics and Automation, pages 2138--2145, 1994.
Latombe1991	J.C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.
Lavalle	S. M. LaValle. Course Notes CS 497, http://msl.cs.uiuc.edu/~lavalle/cs497/, Spring 2002.
Lavalle1998	S. M. LaValle. *Rapidly-exploring random trees: A new tool for path planning*. TR 98-11, Computer Science Dept., Iowa State Univ. <http://janowiec.cs. iastate.edu/papers/rrt.ps>, Oct. 1998.

APPENDIX A – Full Source Code

Graph.h:

```
// ------------------------------------------------------------------------
// Graph.h
//
// Written by Brent M. Dingle, Spring 2002
// ------------------------------------------------------------------------


#ifndef BMD_GRAPH_CLASS

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
#define WORD_NOT_SET              "WordNotSet"
#define MAX_NHBRS                 30
#define MAX_RECURSE_DEPTH         30

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
class CNode
{
public:
   CNode();
   CNode(char *word, long id);
   CNode(const CNode& rhs);          // copy constructor
   ~CNode();
   long AddNhbr(CNode *addme);

   char m_word[25];
   long m_id;
   CNode *m_nhbr[MAX_NHBRS];  // these will NOT be allocated by this class
   long m_num_nhbrs;

private:
};

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
class CNodeList
{
public:
   CNodeList();
   ~CNodeList();

   void SetNode(CNode tothis);
   CNodeList* AddNode(CNode addme);
   CNodeList* FindNode(CNode *findme);
   void RemoveNode(CNode delme);
   long MakeNhbrs(CNode v1, CNode v2);
   bool FindPath(CNode *from, CNode *to, CNodeList *the_path,
                 long recurse_depth = 1, bool check_ends = true);

   void OutputList();
   void OutputListWithNhbrs(long how_many);

   CNode m_node;
   CNodeList *mp_next;

private:
   CNodeList* AddWithJoin(CNode *addme, long dist=1);
   long WordDiff(char *w1, char *w2);
};

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
```

```
class CGraph
{
public:
   CGraph();
   ~CGraph();

   AddNode(CNode addme);
   AddEdge(CNode v1, CNode v2); // adds verts also if not already in graph

   FindNode(CNode findme);

   bool FindPath(CNode *from, CNode *to, CNodeList *the_path,
                 long recurse_depth = 1, bool check_ends = true);

   void OutputListWithNhbrs(long how_many);


   CNodeList m_list;   // edges are by nhbrs of node

};

#define BMD_GRAPH_CLASS
#endif
```

Graph.cpp;

```
// ---------------------------------------------------------------------
// Graph.cpp
//
// Written by Brent M. Dingle, Spring 2002
//
// ---------------------------------------------------------------------

#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <ctype.h>
#include <string.h>

#include "graph.h"


// ---------------------------------------------------------------------
// ---------------------------------------------------------------------
CNode::CNode()
{
   long i;

   strcpy(m_word, WORD_NOT_SET);
   m_id = -1;

   for (i=0; i < MAX_NHBRS; i++)
   {
      m_nhbr[i] = NULL;
   }

   m_num_nhbrs = 0;
}

CNode::CNode(char *word, long id)
{
   long i;

   strcpy(m_word, word);
   m_id = id;

   for (i=0; i < MAX_NHBRS; i++)
   {
      m_nhbr[i] = NULL;
   }

   m_num_nhbrs = 0;
}

CNode::CNode(const CNode& rhs)
{
   long i;

   strcpy(m_word, rhs.m_word);
   m_id = rhs.m_id;
   for (i=0; i < MAX_NHBRS; i++)
   {
      m_nhbr[i] = rhs.m_nhbr[i];   // this may cause bad things to happen
   }
   m_num_nhbrs = rhs.m_num_nhbrs;

}

CNode::~CNode()
{
   // do nothing, nhbrs NOT allocated by this class
}

long CNode::AddNhbr(CNode *addme)
{
   long ret_val;

   ret_val = 0;

   if (m_num_nhbrs < MAX_NHBRS)
   {
      m_nhbr[m_num_nhbrs] = addme;
      m_num_nhbrs++;
      ret_val = m_num_nhbrs;
   }

   return ret_val;
}



// ---------------------------------------------------------------------
// ---------------------------------------------------------------------
// ---------------------------------------------------------------------
// ---------------------------------------------------------------------
// ---------------------------------------------------------------------
// ---------------------------------------------------------------------
CNodeList::CNodeList()
{
   strcpy(m_node.m_word, WORD_NOT_SET);
   mp_next = NULL;
}

// ---------------------------------------------------------------------
CNodeList::~CNodeList()
{
   if (mp_next != NULL)
   {
      delete mp_next;
      mp_next = NULL;
   }
}

// ---------------------------------------------------------------------
// ---------------------------------------------------------------------
void CNodeList::SetNode(CNode tothis)
{
   m_node = tothis;  // notice this COPIES the tothis
}
```

```
// ----------------------------------------------------------------------
// ----------------------------------------------------------------------
CNodeList* CNodeList::AddNode(CNode addme)
{
   CNodeList *ret_ptr;

   ret_ptr = NULL;

   // don't allow duplicate node/words
   if (strcmp(m_node.m_word, addme.m_word)==0)
   {
      return NULL;
   }

   if (strcmp(m_node.m_word, WORD_NOT_SET) == 0) // this is the head node of the list
   {
      m_node = addme;
      ret_ptr = this;
   }
   else
   {
      if (mp_next != NULL)
      {
         ret_ptr = mp_next->AddNode(addme);
      }
      else
      {
         mp_next = new CNodeList;
         mp_next->SetNode(addme);
         mp_next->mp_next = NULL;

         ret_ptr = mp_next;
      }
   }

   return ret_ptr;

} // end AddNode

// ----------------------------------------------------------------------
// AddWithJoin
// dist defaults to 1 (= num chars difference to call words neighbors)
// ----------------------------------------------------------------------
CNodeList* CNodeList::AddWithJoin(CNode *addme, long dist)
{
   CNodeList *ret_ptr;
   CNodeList *ptr;
   long chars_diff;

   ret_ptr = AddNode(*addme);   // ret_ptr thus pts at mem in LIST

   if (ret_ptr != NULL)
   {
      // cycle through all members already in list
      // if any are within 1 to dist characters of addme
      // Make them Neighbors
      ptr = this;
      while (ptr != NULL)
      {
         chars_diff = WordDiff(ptr->m_node.m_word, ret_ptr->m_node.m_word);
         if ((chars_diff >= 1) && (chars_diff <= dist))
         {
            MakeNhbrs(ret_ptr->m_node, ptr->m_node);
         }

         // advance the ptr for looping
         ptr = ptr->mp_next;
         if (ptr == ret_ptr)
         {
            ptr = ptr->mp_next;  // should become NULL
         }
      } // end while ptr != NULL
   } // end if

   return ret_ptr;
} // end AddWithJoin

// ---------------------------------------------------------------------------
// WordDiff
//
// Return number of character different between w1 and w2
// ---------------------------------------------------------------------------
long CNodeList::WordDiff(char *w1, char *w2)
{
   long l1, l2, max;
   long i;
   long diff;

   l1 = strlen(w1);
   l2 = strlen(w2);
   max = l2;           // guess that l1 and l2 are equal or l2 is smaller

   diff = l1 - l2;

   if (diff < 0)  // l1 smaller
   {
      max = l1;
      diff = -diff;  // make diff positive
   }

   for (i=0; i < max; i++)
   {
      if (w1[i] != w2[i])
      {
         diff++;
      }
   } // end for

   return diff;
} // end WordDiff

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
CNodeList* CNodeList::FindNode(CNode *findme)
{
   CNodeList *ret_ptr;

   ret_ptr = NULL;

   if (strcmp(m_node.m_word, findme->m_word) == 0)
   {
      ret_ptr = this;
   }
   else
   {
      if (mp_next != NULL)
      {
         ret_ptr = mp_next->FindNode(findme);
      }
      else
      {
         ret_ptr = NULL;
      }
   }
   return ret_ptr;
} // end FindNode


// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
void CNodeList::RemoveNode(CNode delme)
```

```
{
   CNodeList *find_ptr, *tmp_ptr;
   bool found;

   if (strcmp(m_node.m_word, delme.m_word) == 0)
   {
      if (mp_next == NULL)  // list becomes empty
      {
         strcpy(m_node.m_word, WORD_NOT_SET);
      }
      else  // make this (head) node equal the next and delete the next
      {
         m_node = mp_next->m_node;
         tmp_ptr = mp_next;
         mp_next = tmp_ptr->mp_next;
         delete tmp_ptr;
         tmp_ptr = NULL;
      }
   }
   else
   {
      tmp_ptr = this;
      find_ptr = mp_next;
      found = false;
      while ((find_ptr != NULL) && ( !found) )
      {
         if (strcmp(find_ptr->m_node.m_word, delme.m_word) == 0)
         {
            found = true;
            tmp_ptr->mp_next = find_ptr->mp_next;
            delete find_ptr;
            find_ptr = NULL;
         }
         else
         {
            tmp_ptr = find_ptr;
            find_ptr = find_ptr->mp_next;
         }
      } // end while find_ptr != NULL and not found
   }

} // end RemoveNode


// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
long CNodeList::MakeNhbrs(CNode v1, CNode v2)
{
   CNodeList *v1_node, *v2_node;

   v1_node = FindNode(&v1);
   v2_node = FindNode(&v2);

   if (v1_node == NULL)
   {
      v1_node = AddNode(v1);
      if (v1_node == NULL)
      {
         cout << "ERROR in CNodeList::MakeNhbrs, v1" << endl;
         return 0;
      }
   }

   if (v2_node == NULL)
   {
      v2_node = AddNode(v2);
      if (v2_node == NULL)
      {
         cout << "ERROR in CNodeList::MakeNhbrs, v2" << endl;
         return 0;
      }
   }

   v1_node->m_node.AddNhbr(&(v2_node->m_node));
   v2_node->m_node.AddNhbr(&(v1_node->m_node));

   return 1;
} // end MakeNhbrs


// ----------------------------------------------------------------------
// recurse depth is how deep into the recursion we currently are
// MAX_RECURSE_DEPTH is how far we are allowed to go
// ----------------------------------------------------------------------
bool CNodeList::FindPath(CNode *from, CNode *to, CNodeList *the_path,
                         long recurse_depth, bool check_ends)
{
   CNodeList *start, *end;
   long i;
   bool match;

   if (recurse_depth > MAX_RECURSE_DEPTH)
   {
      return false;
   }

   if (check_ends)
   {
      start = FindNode(from);
      end = FindNode(to);
      if ((start == NULL) || (end == NULL))
      {
         cout << "Unable to locate from or to node" << endl;
         cout << "Attempting to add them and join them to current graph" << endl;
         if (start == NULL)
         {
            start = AddWithJoin(from);  // AddWithJoin is sort of a local planner
         }
         if (end == NULL)
         {
            end = AddWithJoin(to);
         }
// debug check
cout << "New graph list with nhbrs is:" << endl;
OutputListWithNhbrs(5);

      } // end if start or end not found

      from = &(start->m_node);   // this is so the pointers are the nodes
      to = &(end->m_node);       // INSIDE the list guaranteed

      // Add the start node to the PATH
      the_path->AddNode(start->m_node);
// debug check
//cout << "START node added to the path, path is currently:" << endl;
//the_path->OutputList();
//cout << "----------------------------" << endl;

   }

   // Check that to isn't an immediate nhbr of from
   i = 0;
   match = false;
   while ((i < from->m_num_nhbrs) && ( !match))
   {
      if (strcmp(from->m_nhbr[i]->m_word, to->m_word) == 0)
      {
         match = true;
         the_path->AddNode(*(from->m_nhbr[i]));
// debug check
//cout << "Node added to the path, path is currently:" << endl;
//the_path->OutputList();
//cout << "----------------------------" << endl;

      }
      i++;
   } // end while

   // If "to node" was NOT only 1 away from "from node"
   // check all nhbrs (depth first type search) of from node
   i = 0;
   while ((i < from->m_num_nhbrs) && ( !match))
   {
      // prevent circular travel
      if (the_path->FindNode(from->m_nhbr[i]) == NULL)
      {
         the_path->AddNode(*(from->m_nhbr[i]));
         match = FindPath(from->m_nhbr[i], to, the_path,
                          recurse_depth+1, false);
         if (!match)
         {
            the_path->RemoveNode(*(from->m_nhbr[i]));
         }
         else
         {
// debug check
//cout << "Node added to the path, path is currently:" << endl;
//the_path->OutputList();
//cout << "----------------------------" << endl;
         }
      } // end if from->nhbr[i] is NOT currently in the_path

      i++;
   } // end while

   return match;
} // end FindPath


// ----------------------------------------------------------------------
// ----------------------------------------------------------------------
void CNodeList::OutputList()
{
   CNodeList *ptr;

   if (strcmp(m_node.m_word, WORD_NOT_SET) != 0) // this is the head node of the list
   {
      cout << m_node.m_word << endl;
   }

   ptr = mp_next;

   while (ptr != NULL)
   {
      cout << ptr->m_node.m_word << endl;
      ptr = ptr->mp_next;
   }
} // end OutputList

// ----------------------------------------------------------------------
// OutputListWithNhbrs
//
// Output list showing first how_many neighbors of each node
// ----------------------------------------------------------------------
void CNodeList::OutputListWithNhbrs(long how_many)
{
   CNodeList *ptr;
   long i;

   if (strcmp(m_node.m_word, WORD_NOT_SET) != 0) // this is the head node of the list
   {
      cout << m_node.m_word << endl;
   }
   ptr = mp_next;

   while (ptr != NULL)
   {
      cout << ptr->m_node.m_word;

      // output the nhbrs of current node
      i = 0;
      while ((i < ptr->m_node.m_num_nhbrs) && (i < how_many) && (i < MAX_NHBRS))
      {
         cout << " - " << ptr->m_node.m_nhbr[i]->m_word;

         i++;
      } // end while

      cout << endl;

      ptr = ptr->mp_next;
   }
} // end OutputListWithNhbrs



// -----------------------------------------------------------------------
// -----------------------------------------------------------------------
// -----------------------------------------------------------------------
// -----------------------------------------------------------------------
// -----------------------------------------------------------------------
// -----------------------------------------------------------------------
CGraph::CGraph()
{
}

CGraph::~CGraph()
{
}

CGraph::AddNode(CNode addme)
{
   m_list.AddNode(addme);
}

// adds verts also if not already in graph
CGraph::AddEdge(CNode v1, CNode v2)
{
   m_list.MakeNhbrs(v1, v2);
}

CGraph::FindNode(CNode findme)
{
   bool ret_val;

   ret_val = false;

   if (m_list.FindNode(&findme) != NULL)
   {
      ret_val = true;
   }

   return ret_val;
}

bool CGraph::FindPath(CNode *from, CNode *to, CNodeList *the_path,
                      long recurse_depth, bool check_ends)
{
   bool ret_val;

   // may add some simple path planning here, in case
   // from and to are not already in the graph
   // For example: if not then see if any of the distance =1 nhbrs are
   // Or simply add them into the graph and then add all edges to their
   // distance 1 nhbrs

   ret_val = m_list.FindPath(from, to, the_path, recurse_depth, check_ends);

   return ret_val;
}

// -----------------------------------------------------------------------
// OutputListWithNhbrs
// -----------------------------------------------------------------------
void CGraph::OutputListWithNhbrs(long how_many)
{
   m_list.OutputListWithNhbrs(how_many);
}
```

PlanPath.cpp:

```
// -----------------------------------------------------------------------
// PlanPath.cpp
// Written By Brent Dingle
//
// This program will create a graph to be used to solve word ladder
// problems.
// The user must supply a the number of letters in the word
// to create a path for. The corresponding word file will be used to
// create the path. (e.g. 4 letter words will require g_word4.txt
// to be used).
//
// The user may also specify the desired maximum degree of each
// vertex in the solution graph (the default is 10).
//
// The solution path graph will be output to path[i].txt
// where [i] = number of letters (e.g four letters means output file
// will be named path4.txt)
//
// -----------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <iostream.h>
#include <ctype.h>
#include <string.h>
#include <math.h>

#include "graph.h"


// -----------------------------------------------------------------------
//                                                         DEFINES
// -----------------------------------------------------------------------
// below are needed for the Random Number Function
#define MBIG    1000000000
#define MSEED    161803398
#define MZ      0
#define FAC ((float)1.0 / MBIG)

// Now we get on with more 'significant' defines
#define DEFAULT_MAX_DEG            10
#define MAX_LETS                   25

// Set MakeGraph for why increasing FRACT_OF_WORDS decreases likelihood of success
// (decreases sample size)
#define FRACT_OF_WORDS              4     // originally 4

// -----------------------------------------------------------------------
//                                                         PROTOTYPES
// -----------------------------------------------------------------------
void DisplayHelp(void);
```

```
long RandomLong(long max_num, long *seed);
float Ran3(long *idum);
void Num2Str(char *str, long num);
long GetWord(FILE *infile, char *word);

void MakeGraph(long num_lets, long max_deg, CGraph *graph);


void PRM(CGraph *graph, char *inname, long words_in_file,
         long num_lets, long num_verts, long max_deg);
void GetRandomNode(FILE *infile, long max_index, CNode *node, CGraph *graph);
void CheckNhbrs(FILE *infile, long num_lets,
                CGraph *graph, CNode *node, long *num_verts);
long FindWord(FILE *infile, char *word);

// ----------------------------------------------------------------------
//        MAIN                    MAIN                     MAIN
// ----------------------------------------------------------------------
int main(int argc, char *argv[])
{
   CNodeList path;
   CNode  node1, node2;
   CGraph graph;
   long num_lets, max_deg;

   cout << "Plan Path for word ladder problems." << endl;
   cout << "Program written by Brent Dingle, Spring 2002" << endl;
   cout << "All rights reserved, no warranty expressed or implied." << endl;
   cout << "Use at your own risk." << endl;
   cout << endl;

   if (argc < 2)
   {
      DisplayHelp();
      exit(0);
   }

   // Set number of letters in the words to solve for
   num_lets = atol(argv[1]);
   if ((num_lets < 4) || (num_lets > MAX_LETS))
   {
      cout << "Number of letters must be from 4 to " << MAX_LETS << "." << endl;
      cout << "Program ends." << endl;
      exit(0);
   }

   // Set the maximum degree (number of connected neighbors to any vertex)
   if (argc > 2)
   {
      max_deg = atol(argv[2]);
      if ((max_deg < 2) || (max_deg > 50))
      {
         cout << "Maximum degree must be from 2 to 50." << endl;
         cout << "Program ends." << endl;
         exit(0);
      }
   }
   else
   {
      max_deg = DEFAULT_MAX_DEG;
   }

   MakeGraph(num_lets, max_deg, &graph);

   cout << "Graph creation complete." << endl << endl;

//     strcpy(node1.m_word, "cats");
//     strcpy(node2.m_word, "hush");
     strcpy(node1.m_word, "wise");
     strcpy(node2.m_word, "fool");
   // do NOT manually add the start and end nodes into the graph!!!!
   // FindPath will do it when needed.

   // See if we can find a path from node1 to node2
   if ( !graph.FindPath(&node1, &node2, &path))
   {
      cout << "Path from " << node1.m_word << " to " << node2.m_word << " not found." <<
endl;
   }
   else
   {
      cout << "Path found:" << endl;
      path.OutputList();
   }

   cout << "Program ends." << endl;

   cout << "Press any key." << endl;
   getch();

   return 0;
} // end main

// ----------------------------------------------------------------------
// DisplayHelp                                              DisplayHelp
// ----------------------------------------------------------------------
void DisplayHelp()
{
   cout << endl;
   cout << "Directions:" << endl;
   cout << "------------------------------------------------------------" << endl;
   cout << "planpath [num_letters]   <max_degree>" << endl;
   cout << endl;
   cout << "[num_letters] = number of letters in words of" << endl;
   cout << "                 the problem to be solved." << endl;
   cout << "<max_degree> = optional parameter to declare the maximum" << endl;
   cout << "                number of nhbrs to use in the path graph for" << endl;
   cout << "                each vertex." << endl;
   cout << endl;
   cout << "Also g_word[nnn].txt files should be in the same directory" << endl;
   cout << "as this executable, where nnn = 001, 002, ... 999" << endl;
   cout << "as [num_letters] dictates." << endl;
   cout << endl;
} // end DisplayHelp

// ----------------------------------------------------------------------
// RandomLong
//
// Return an integer from 0 to max_num
// ----------------------------------------------------------------------
long RandomLong(long max_num, long *seed)
{
   float num;
   long ret_val;

   num = Ran3(seed);
   num *= max_num;

   ret_val = (long)num;

   return ret_val;
} // end RandomLong

// ----------------------------------------------------------------------
// Rans3   -- subtractive method for Uniform random number
//
// Returns a UNIFORM random deviate between 0.0 and 1.0.
// Set idum to any negative value to initialize or
// reinitialiize the sequence.
//
```

```
// ----------------------------------------------------------------------
float Ran3(long *idum)
{
   static int inext, inextp;
   static long ma[56];           // 56 should NOT be changed
   static int iff = 0;

   long mj, mk;
   int i, ii, k;

   // Initialization
   if ( (*idum < 0) || (iff == 0))
   {
      iff = 1;

      // Init ma[55] using seed idum and MSEED
      mj = labs(MSEED - labs(*idum));
      mj %= MBIG;
      ma[55] = mj;
      mk = 1;

      // Initialize the rest of the table in slightly random order
      // with numbers that are not especially random
      for (i = 1; i <= 54; i++)
      {
         ii = (21*i) % 55;
         ma[ii] = mk;
         mk = mj - mk;
         if (mk < MZ)
         {
            mk += MBIG;
         }
         mj = ma[ii];
      }

      // Randomize the items by 'warming up' the generator
      for (k = 1; k <= 4; k++)
      {
         for (i = 1; i <= 55; i++)
         {
            ma[i] -= ma[ 1 + (i+30) % 55];
            if (ma[i] < MZ)
            {
               ma[i] += MBIG;
            }
         } // end for i
      } // end for k
      inext = 0;                 // prep indices for first generated number
      inextp = 31;               // 31 is SPECIAL
      *idum = 1;
   } // end initialization

   // Increment inext and inextp, with wrap from 56 to 1
   if (++inext == 56)
   {
      inext = 1;
   }
   if (++inextp == 56)
   {
      inextp = 1;
   }

   // Generate new random number subtractively
   mj = ma[inext] - ma[inextp];

   // Check that it is in range
   if (mj < MZ)
   {
      mj += MBIG;
   }
   // store it
   ma[inext] = mj;

   // output derived uniform deviate
   return mj*FAC;

} // end Ran3

// -----------------------------------------------------------------------
// Num2Str                                                 Num2Str
//
// Convert a given number to a string of 3 characters
// e.g. 1 becomes "001"
//
// -----------------------------------------------------------------------
void Num2Str(char *str, long num)
{
   char tmp_str[10];

   ltoa(num, tmp_str, 10);  // value, string radix

   if (num < 10)  // add 2 zeros
   {
      strcpy(str, "00");
   }
   else if (num < 100)  // add 1 zero
   {
      strcpy(str, "0");
   }
   else
   {
      str[0] = '\0';
   }

   strcat(str, tmp_str);

} // end Num2Str

// -----------------------------------------------------------------------
// GetWord                                                 GetWord
// -----------------------------------------------------------------------
long GetWord(FILE *infile, char *word)
{
   long length;
   int ch;

   length = 0;

   ch = fgetc(infile);
   ch = tolower(ch);

   while ( ( !feof(infile) ) &&
           ( (isalpha(ch)) || (ch == '-') || (ch == 39) ) // chr(39) = apostrophe
         )
   {
      word[length] = ch;
      length++;

      ch = fgetc(infile);
      ch = tolower(ch);

      // failsafe - do this better
      if (length > 35)
      {
         break;
      }
   } // end while

   word[length] = '\0';

   return length;
```

```
} // end GetWord

// ----------------------------------------------------------------------
// MakeGraph
// ----------------------------------------------------------------------
void MakeGraph(long num_lets, long max_deg, CGraph *graph)
{
   FILE *infile;
   char inname[30], num_str[10];
   char tmp_word[30];
   long words_in_file, length;
   long num_verts;

   Num2Str(num_str, num_lets);
   strcpy(inname, "g_word");
   strcat(inname, num_str);
   strcat(inname, ".txt");
   infile = fopen(inname, "r");
   if (infile == NULL)
   {
      cout << "Error -- Unable to open file: [" << inname << "]" << endl;
      return;
   }
   cout << "Reading from file: " << inname << endl;
   cout << "   Processing words of length = " << num_lets << endl;

   words_in_file = 0;
   while (!feof(infile))
   {
      length = GetWord(infile, tmp_word);
      if (length == num_lets)
      {
         words_in_file++;
      }
   }

   cout << "   There are " << words_in_file << " words in the known configuration space."
<< endl;
   cout << endl;

   // reset to the beginning of the file - this is the dumb way
   fclose(infile);

   if (words_in_file > 10)
   {
      num_verts = words_in_file / FRACT_OF_WORDS;   // use only 1/4 of the possible nodes
      PRM(graph, inname, words_in_file, num_lets, num_verts, max_deg);
   }
   else
   {
      cout << "There are only 1 or 2 words in the dictionary." << endl;
      cout << "Even a human should be able to process that." << endl;
      cout << endl;
   }

} // end MakeGraph

// ----------------------------------------------------------------------
// PRM - Probablistic Roadmap General Method
//
// Create and save a graph for potentially solving a word ladder puzzle.
//
// inname is assumed to be valid and NOT open AND MUST contain
// [words_in_file] words.
// ----------------------------------------------------------------------
void PRM(CGraph *graph, char *inname, long words_in_file,
         long num_lets, long num_verts, long max_deg)
{
   FILE   *infile;
   CNode  node;
   long i;

   infile = fopen(inname, "r");

   // Graph is Initialized to contain zero vertices
   // When we are done it will contain num_verts
   // howevr, not all may be connected

   i = 1;
   while (i <= num_verts)  // may alter i outside this func so do NOT use a for loop
   {
      GetRandomNode(infile, words_in_file, &node, graph);
      graph->AddNode(node);
      i++;
// debug check:
cout << "Random node = " << node.m_word << "... num verts so far is " << i << endl;

      // For each node v (valid word), whose distance is one from node
      // add edge from v to node - also add v into graph if not already in graph
      // (and increment i so our vertex count is correct)
      CheckNhbrs(infile, num_lets, graph, &node, &i);

   } // end while i

   fclose(infile);
} // end PRM

// ----------------------------------------------------------------------
// GetRandomNode
// infile is assumed opened already
// ----------------------------------------------------------------------
void GetRandomNode(FILE *infile, long max_index, CNode *node, CGraph *graph)
{
   static long seed = -3; // change this to some other negative num for diff results
   long rand_index;
   long index;
   char word[30], user_in[10];
   bool valid;
   long num_tries;

   rand_index = RandomLong(max_index-1, &seed);

   num_tries = 1;
   valid = false;
   while (!valid)
   {
      // technically if we use binary read and fseek we can do this more efficiently
      rewind(infile);
      for (index=1; index <= rand_index; index++)
      {
         GetWord(infile, word);
      }

      // Set the node
      strcpy(node->m_word, word);
      node->m_id = rand_index;        // id's are not really used (yet)

      // check if word is valid (i.e. not already in the graph)
      if (graph->FindNode(*node) == false)
      {
         valid = true;
      }
      else // pick another random number
      {
         rand_index = RandomLong(max_index-1, &seed);
         num_tries++;

         if (num_tries % 5000 == 0)
         {
            cout << "Stuck in GetRandomNode, num_tries = " << num_tries << endl;
            cout << "Do you wish to continue trying? " << endl;
            gets(user_in);
            if ((user_in[0] == 'n') || (user_in[0] == 'N'))
            {
               cout << " Program terminated. " << endl;
               fclose(infile);
               exit(1);
            }
         } // end asking user ifwish to continue

      } // end else

   } // end while not valis
} // end GetRandomNode


// ----------------------------------------------------------------------
// CheckNhbrs
// For each node v (valid word), whose distance is one from node
// add edge from v to node - also add v into graph if not already in graph
// (and increment i so our vertex count is correct)
//
// infile is assumed opened and its pointer will be changed
//
// ----------------------------------------------------------------------
void CheckNhbrs(FILE *infile, long num_lets,
                CGraph *graph, CNode *node, long *num_verts)
{
   CNode nhbr_node;
   long let_index;
   char ch;

   // loop through each character of node's word
   for (let_index = 0; let_index < num_lets; let_index++)
   {
      strcpy(nhbr_node.m_word, node->m_word);

      // and change the letter
      for (ch = 'a'; ch <= 'z'; ch++)
      {
         nhbr_node.m_word[let_index] = ch;

         if (node->m_word[let_index] != ch)
         {
            // if nhbr_word is a 'known' word -- shows up in infile
            nhbr_node.m_id = FindWord(infile, nhbr_node.m_word);
            if (nhbr_node.m_id >= 0)
            {
               if (graph->FindNode(nhbr_node) == false)
               {
                  graph->AddNode(nhbr_node);   // notice could recurse this function on
this node's nhbrs too
                  *num_verts = *num_verts + 1;
               }
               graph->AddEdge(*node, nhbr_node);
            } // end word was valid
         }
      } // end for ch
   } // end for let_index

} // end CheckNhbrs


// ----------------------------------------------------------------------
// FindWord
//
// Returns the word number of word, if it is found in infile
// else returns negative 1
//
// ----------------------------------------------------------------------
long FindWord(FILE *infile, char *word)
{
   long ret_val, index;
   char tmp_word[30];

   ret_val = -1;
   rewind(infile);

   tmp_word[0] = '\0';   // as a precaution;

   index = 0;
   while ( ( !feof(infile)) && (ret_val == -1) )
   {
      GetWord(infile, tmp_word);

      if (strcmp(word, tmp_word) == 0)
      {
         ret_val = index;
      }
      index++;
   } // end while

   return ret_val;

} // end FindWord

// -----------------------------------------------------------------------
// -----------------------------------------------------------------------
// end PlanPath.cpp
```