Tristan Shores

ECE478

Sunday, November 5, 2012

Search Strategy Exploration

SCOPE OF ASSIGNMENT

Two independent simulated search-space challenges were explored:

A. Identifying unknown molecular compounds by their atomic mass. A knowledge of chemistry was used to determine the possible combinations of elements whose individual mass adds up to the compound's atomic mass.
B. Discovering the shortest path from a start point to an endpoint on the first floors of the Maseeh Building and Cramer Hall.

Each challenge will be discussed separately below.

IDENTIFICATION OF A MOLECULAR COMPOUND

Introduction

An expert system (standalone C# program) was implemented to determine the possible combinations of elements whose individual mass makes up a compound's atomic mass. Heuristics, based on highly simplified rules for covalent/ionic bonding, were applied on top of breadth-first, depth-first, and iterative depth-first searches to narrow search scope by limiting generation of child nodes.

Following an in-class presentation of the expert system the following modifications were made:

1. Final solutions were formatted into chemical symbols using Hill notation.
2. A database of common chemical compounds was used to indicate which of the final solutions was known. However, that database was not used to limit the candidate set because that would conflict with one of the goals of the expert system: to identify unknown and/or unstable compounds, which may exist under atypical temperature/pressure conditions.

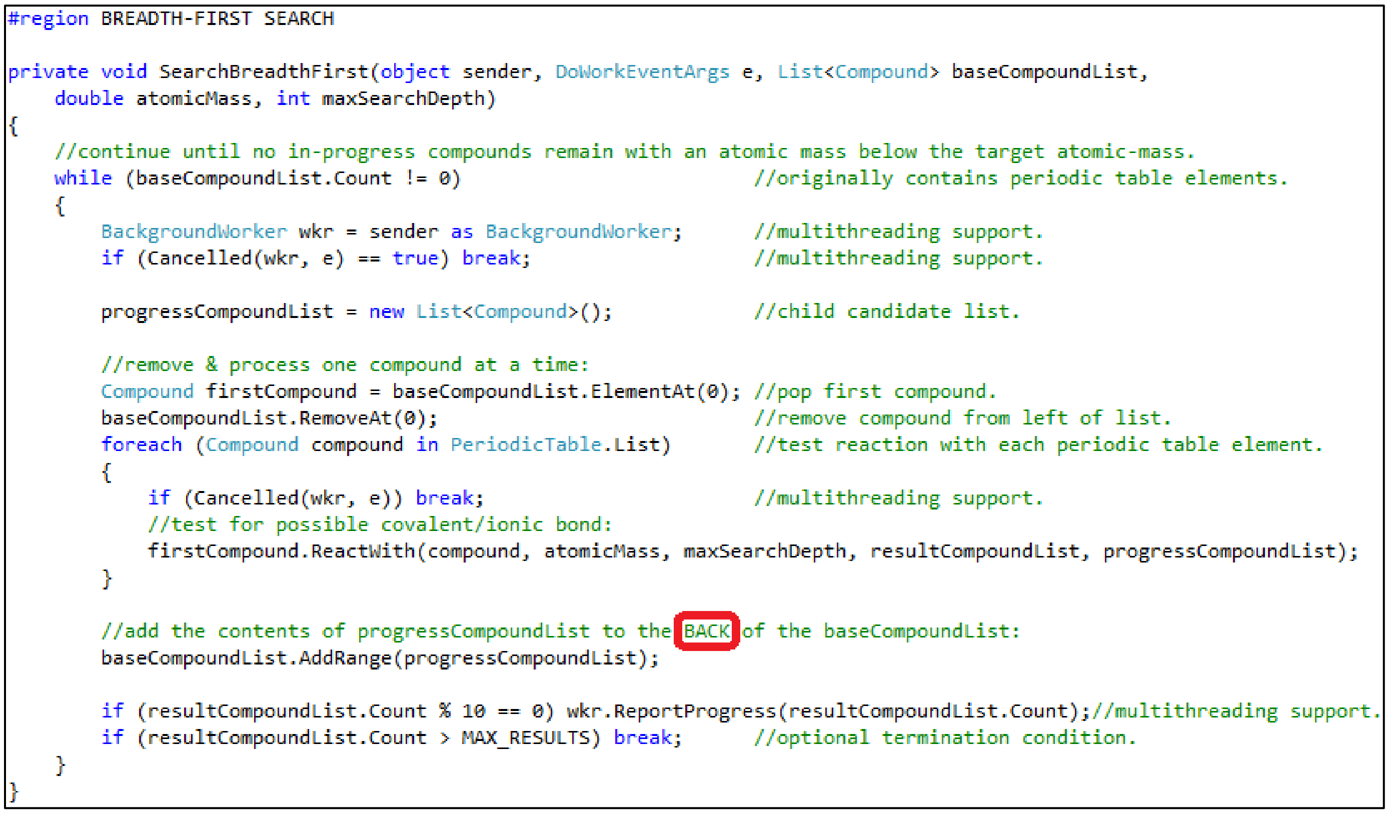
Search Path Strategies

The program allows one of three possible search path strategies to be selected: breadth-first, depth-first, and iterative depth-first. Elements from the periodic table comprise a base set of 65 elements whose various combinations form numerous compounds that exhibit a target atomic mass. The size of the search space with search depth (D) is 65^D, where D also indicates the number of compound elements at that depth. Thus the size of the search space that includes vinegar ($C_2H_4O_2$) is 73^8 ($\approx 10^{15}$). Given the exponential magnitude of the search space a stringent selection criteria (by application of heuristics) is needed to aggressively limit child node candidates. Nevertheless, the program can generate thousands of theoretically possible solutions in a case where the target atomic mass exceeds 100 and the search depth exceeds 5 levels. Since the search space expands exponentially with depth, a search depth limit condition (for each search strategy) was used to restrict the search space.

Breadth-First Search:

The breadth-first search path strategy fully searches each level of the search space before proceeding to the next level. This laterally priority search strategy is suitable when an exhaustive search is required to identify every possible solution that meets the search criteria, or where the solution is located at a shallow node depth.

The program code for the breadth-first search is shown below.

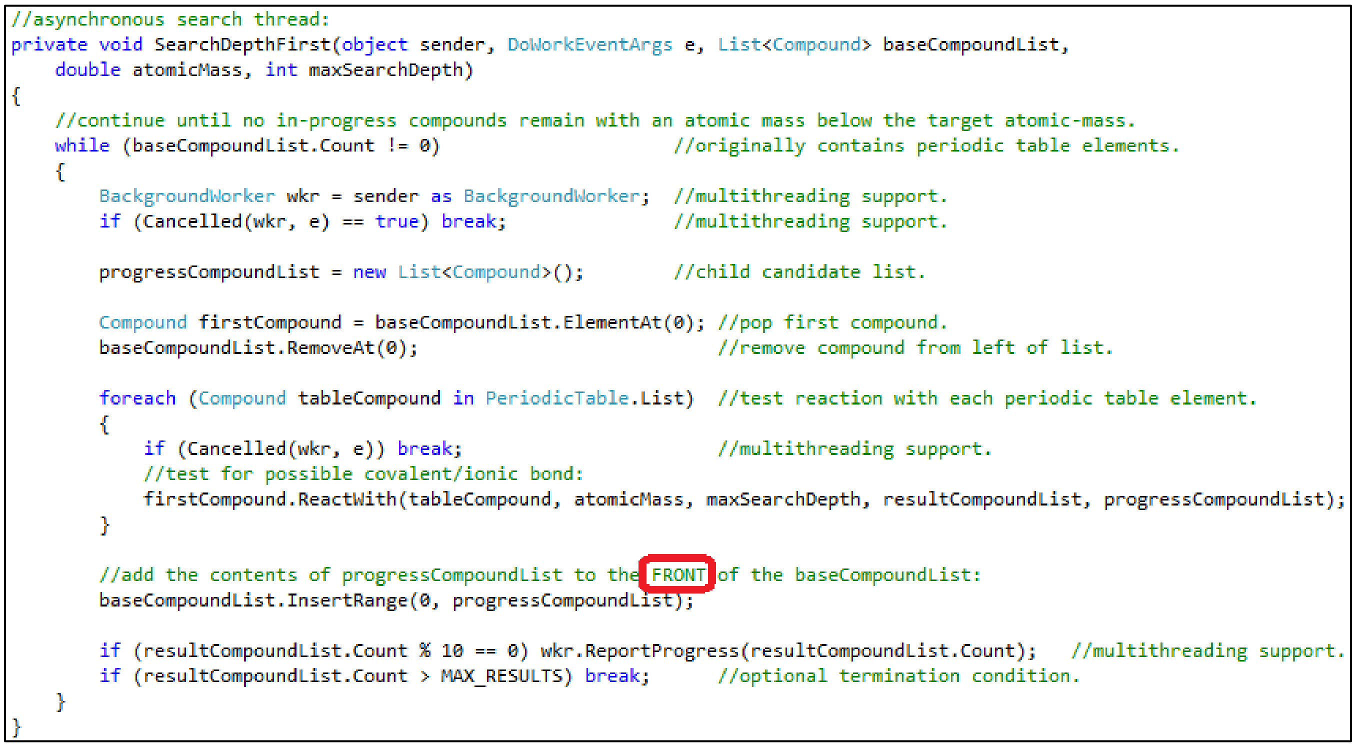


```
#region BREADTH-FIRST SEARCH

private void SearchBreadthFirst(object sender, DoWorkEventArgs e, List<Compound> baseCompoundList,
    double atomicMass, int maxSearchDepth)
{
    //continue until no in-progress compounds remain with an atomic mass below the target atomic-mass.
    while (baseCompoundList.Count != 0)                     //originally contains periodic table elements.
    {
        BackgroundWorker wkr = sender as BackgroundWorker;  //multithreading support.
        if (Cancelled(wkr, e) == true) break;               //multithreading support.

        progressCompoundList = new List<Compound>();        //child candidate list.

        //remove & process one compound at a time:
        Compound firstCompound = baseCompoundList.ElementAt(0); //pop first compound.
        baseCompoundList.RemoveAt(0);                       //remove compound from left of list.
        foreach (Compound compound in PeriodicTable.List)   //test reaction with each periodic table element.
        {
            if (Cancelled(wkr, e)) break;                   //multithreading support.
            //test for possible covalent/ionic bond:
            firstCompound.ReactWith(compound, atomicMass, maxSearchDepth, resultCompoundList, progressCompoundList);
        }

        //add the contents of progressCompoundList to the BACK of the baseCompoundList:
        baseCompoundList.AddRange(progressCompoundList);

        if (resultCompoundList.Count % 10 == 0) wkr.ReportProgress(resultCompoundList.Count);//multithreading support.
        if (resultCompoundList.Count > MAX_RESULTS) break;  //optional termination condition.
    }
}
```

Depth-First Search:

The depth-first search path strategy explores a continuous deepening search path to maximum depth before restarting at the next top level lateral node. This depth prioritized search strategy is suitable where the solution is located at a deep node depth or any solution that meets the criteria is acceptable.

The program code for the depth-first search is shown below.

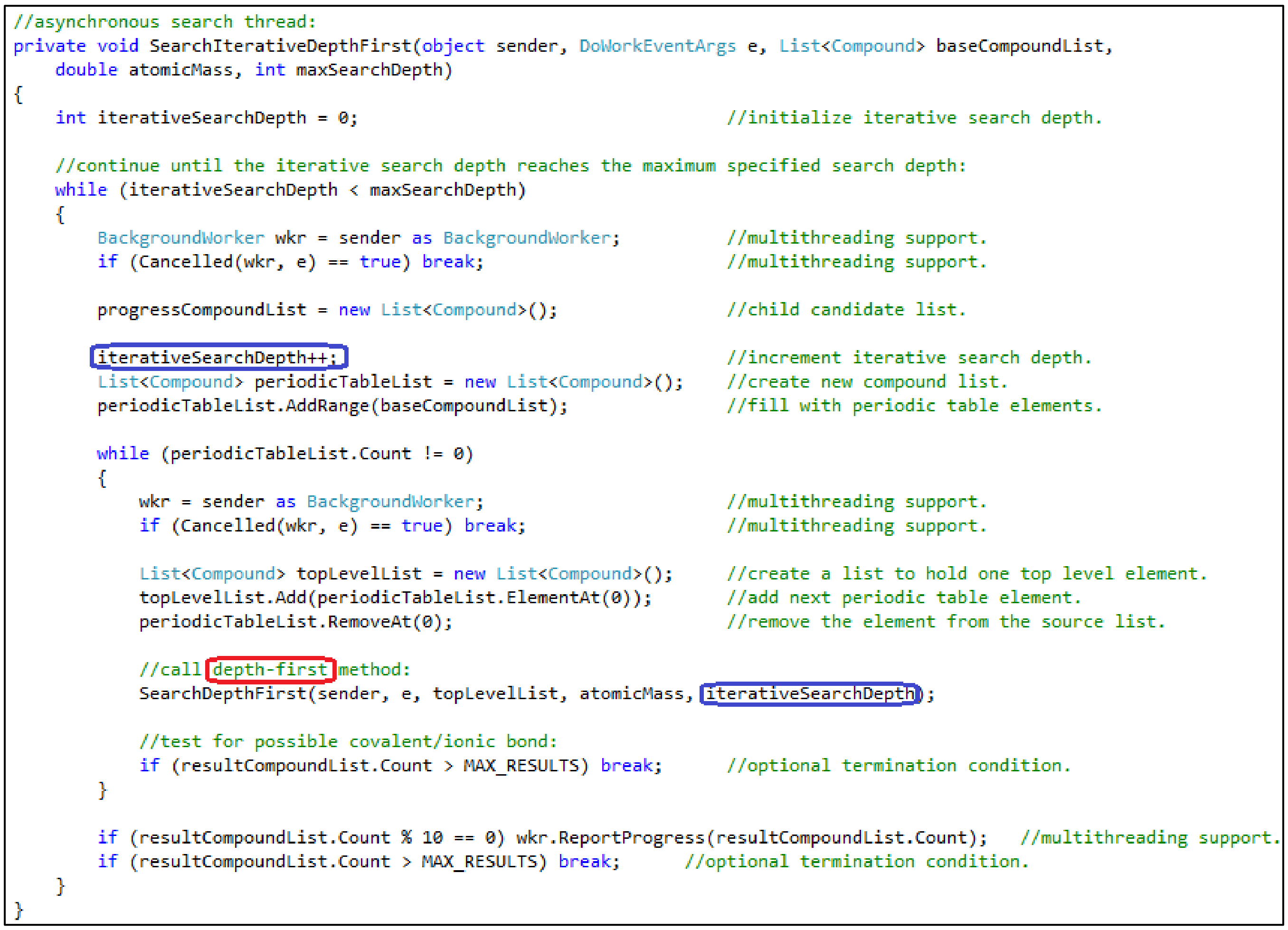


```
//asynchronous search thread:
private void SearchDepthFirst(object sender, DoWorkEventArgs e, List<Compound> baseCompoundList,
    double atomicMass, int maxSearchDepth)
{
    //continue until no in-progress compounds remain with an atomic mass below the target atomic-mass.
    while (baseCompoundList.Count != 0)                     //originally contains periodic table elements.
    {
        BackgroundWorker wkr = sender as BackgroundWorker;  //multithreading support.
        if (Cancelled(wkr, e) == true) break;               //multithreading support.

        progressCompoundList = new List<Compound>();        //child candidate list.

        Compound firstCompound = baseCompoundList.ElementAt(0); //pop first compound.
        baseCompoundList.RemoveAt(0);                           //remove compound from left of list.

        foreach (Compound tableCompound in PeriodicTable.List)  //test reaction with each periodic table element.
        {
            if (Cancelled(wkr, e)) break;                       //multithreading support.
            //test for possible covalent/ionic bond:
            firstCompound.ReactWith(tableCompound, atomicMass, maxSearchDepth, resultCompoundList, progressCompoundList);
        }

        //add the contents of progressCompoundList to the FRONT of the baseCompoundList:
        baseCompoundList.InsertRange(0, progressCompoundList);

        if (resultCompoundList.Count % 10 == 0) wkr.ReportProgress(resultCompoundList.Count);   //multithreading support.
        if (resultCompoundList.Count > MAX_RESULTS) break;      //optional termination condition.
    }
}
```

Iterative Depth-First Search:

The iterative depth-first search path strategy explores iteratively deepening depth-first search paths for each lateral top-level node. The search to a specific node depth is performed exhaustively for successive top level nodes before the next iteration with a deeper depth begins. This search strategy is suitable where the solution is located at an unknown node depth or any solution that meets the criteria is acceptable.

The program code for the depth-first search is shown below.



```
//asynchronous search thread:
private void SearchIterativeDepthFirst(object sender, DoWorkEventArgs e, List<Compound> baseCompoundList,
    double atomicMass, int maxSearchDepth)
{
    int iterativeSearchDepth = 0;                                   //initialize iterative search depth.

    //continue until the iterative search depth reaches the maximum specified search depth:
    while (iterativeSearchDepth < maxSearchDepth)
    {
        BackgroundWorker wkr = sender as BackgroundWorker;          //multithreading support.
        if (Cancelled(wkr, e) == true) break;                      //multithreading support.

        progressCompoundList = new List<Compound>();                //child candidate list.

        iterativeSearchDepth++;                                     //increment iterative search depth.
        List<Compound> periodicTableList = new List<Compound>();    //create new compound list.
        periodicTableList.AddRange(baseCompoundList);               //fill with periodic table elements.

        while (periodicTableList.Count != 0)
        {
            wkr = sender as BackgroundWorker;                       //multithreading support.
            if (Cancelled(wkr, e) == true) break;                  //multithreading support.

            List<Compound> topLevelList = new List<Compound>();     //create a list to hold one top level element.
            topLevelList.Add(periodicTableList.ElementAt(0));       //add next periodic table element.
            periodicTableList.RemoveAt(0);                          //remove the element from the source list.

            //call depth-first method:
            SearchDepthFirst(sender, e, topLevelList, atomicMass, iterativeSearchDepth);

            //test for possible covalent/ionic bond:
            if (resultCompoundList.Count > MAX_RESULTS) break;      //optional termination condition.
        }

        if (resultCompoundList.Count % 10 == 0) wkr.ReportProgress(resultCompoundList.Count);   //multithreading support.
        if (resultCompoundList.Count > MAX_RESULTS) break;      //optional termination condition.
    }
}
```

Heuristics Used for Selection of Child Candidates

A child node (Cc) is generated from a parent compound/element (C1) reacted with one of the elements (C2) from the periodic table when ALL of the following apply:

1. (Atomic mass Cc) $\leq$ (target atomic mass $\pm$ small tolerance).
2. (Total elements that comprise C1 and C2) $\leq$ (target element count)
3. A valid covalent/ionic bond occurs between C1 and C2. Valid bond conditions are:
    a. For a covalent bond:
        i. Both C1 and C2 contain only non-metal elements.
        ii. ((Atomic mass Cc $<$ target atomic mass) AND net charge is not zero) OR ((Atomic mass Cc $=$ target atomic mass) AND net charge is zero).

    Note that the net charge is determined by successively attempting to form a single/double/triple covalent bond between C1 and C2. That bond is informed by the number of valence electrons in the outer shell of C1 and C2. Since a single covalent bond between C1 and C2 forms a different compound than a double/triple covalent bond, Cc can actually be a set of compounds.

   b. For an ionic bond:
      i. C1 solely comprises non-metals and C2 is a metal. Let the result be C3, OR C3 and the C2 element repeated.
      ii. ((Atomic mass Cc) = (target atomic mass) AND net charge is zero) OR ((Atomic mass Cc = target atomic mass) AND net charge is zero).

The resulting child node(s) were placed in one of two lists:

1. In-progress list: when Cc net charge is non-zero.
2. Completed List: when Cc net charge is zero (i.e. fully formed compound with the target atomic mass).

The program code for the heuristics as described above is shown below.

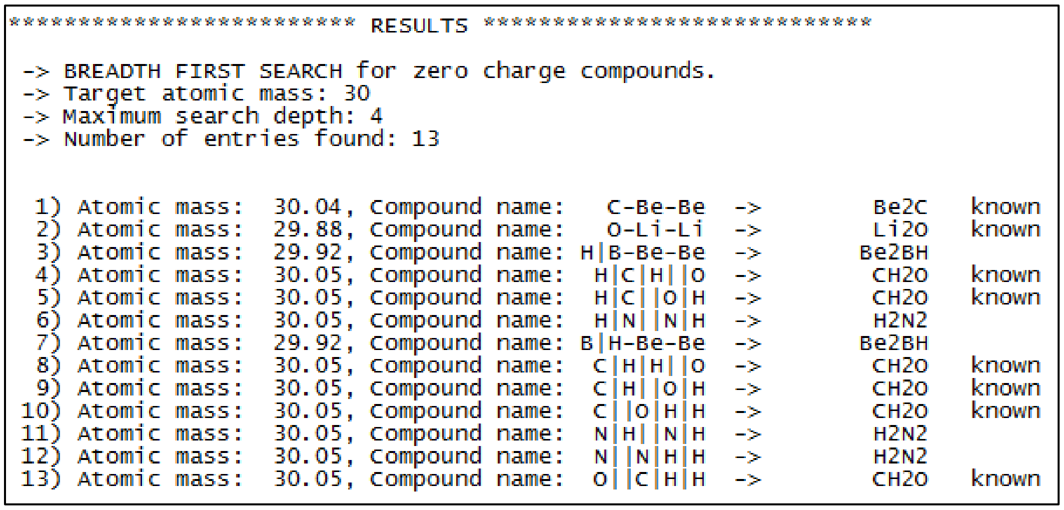
```
public ReactResult ReactWith(Compound other, double targetMass, int targetElementCount,
    List<Compound> completeCompoundList, List<Compound> progressCompoundList)
{
    //check mass of combined compound:
    double newAtomicMass = this.atomicMass + other.atomicMass;
    if (newAtomicMass > targetMass + ERROR) return ReactResult.None;

    //check element count of new compound:
    int newElementCount = this.elementCount + 1;
    if (newElementCount > targetElementCount) return ReactResult.None;

    //only allow covalent bonds between non-metals:
    if (this.metalType == MetalType.NonMetal && other.metalType == MetalType.NonMetal) bondType = BondType.Covalent;
    //allow an ionic bond between a non-metal and a metal:
    else if (this.metalType == MetalType.NonMetal && other.metalType == MetalType.Metal) bondType = BondType.Ionic;
    //allow an ionic bond between a part-metal (with a final metal element) and the same final metal element:
    else if (this.metalType == MetalType.Mixed && other.metalType == MetalType.Metal
        && GetLastCompoundName(this.name) == other.name) bondType = BondType.Ionic;
    else return ReactResult.None;

    //handle covalent bond:
    if (bondType == BondType.Covalent)
    {
        ReactResult reactResult = ReactResult.None;
        string bars = "";
        int commonValenceOffset = Math.Min(Math.Abs(this.valenceOffset), Math.Abs(other.valenceOffset));
        Compound[] compounds = new Compound[commonValenceOffset];
        for (int i = 0; i < commonValenceOffset; i++)
        {
            bars += "|";
            string newName = this.name + bars + other.name;
            int newValenceOffset = this.valenceOffset + other.valenceOffset + 2 * (i + 1);
            compounds[i] = new Compound(newName, newValenceOffset, newAtomicMass, other.electroNegativity,
                newElementCount, MetalType.NonMetal, BondType.Covalent);
            if (newValenceOffset != 0)
            {
                progressCompoundList.Add(compounds[i]);
                reactResult = ReactResult.ProgressCompound;
            }
            else if (newAtomicMass > targetMass - ERROR)
            {
                completeCompoundList.Add(compounds[i]);
                if (reactResult != ReactResult.ProgressCompound) reactResult = ReactResult.CompleteCompound;
            }
            else reactResult = ReactResult.None;    //complete compound, except that atomic mass is too low.
        }
        return reactResult;
    }
    //handle ionic bond:
    else if (bondType == BondType.Ionic)
    {
        int newValenceOffset = this.valenceOffset + other.valenceOffset;
        Compound compound = new Compound(this.name + "-" + other.name, newValenceOffset, newAtomicMass,
            other.electroNegativity, newElementCount, MetalType.Mixed, BondType.Ionic);
        if (newValenceOffset == 0 && newAtomicMass > targetMass - ERROR)
        {
            completeCompoundList.Add(compound);
            return ReactResult.CompleteCompound;
        }
        else if (newValenceOffset < 0)
        {
            progressCompoundList.Add(compound);
            return ReactResult.ProgressCompound;
        }
        else return ReactResult.None;   //new valency value invalid.
    }
    //handle no bond:
    else return ReactResult.None;
}
```

Expert System Search Results

A search for a target atomic mass of 30 with a limit to the maximum search depth of 4 is shown below. This maximum search depth limits the maximum number of elements in the solution set to 4. The output for each search strategy is shown below.

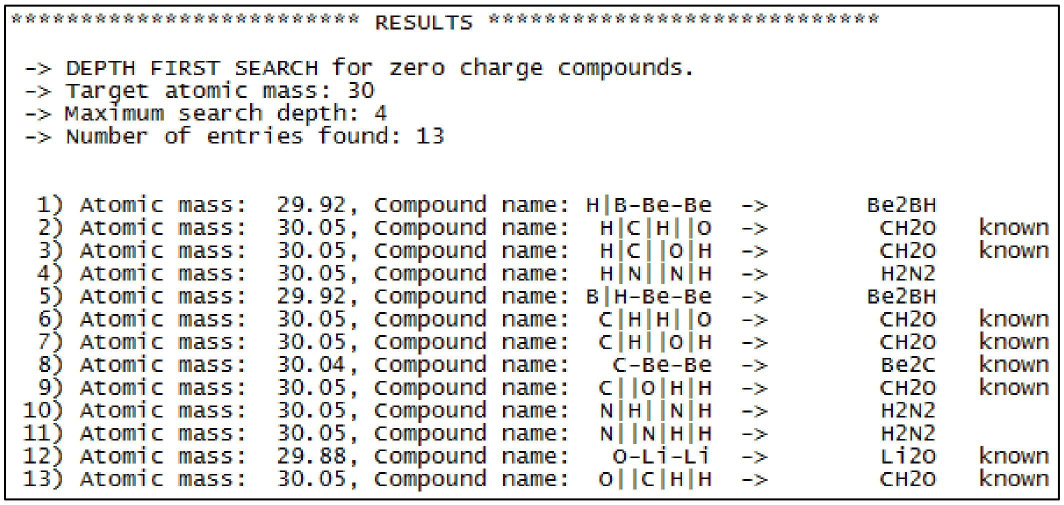
Note that all three search strategies produced the same set of unique solutions, however the iterative depth search has two duplicates due to iterative traversing of the same search space. Note that the ordering of the search results predictably varies according to the particular searching strategy.

Breadth-first results:



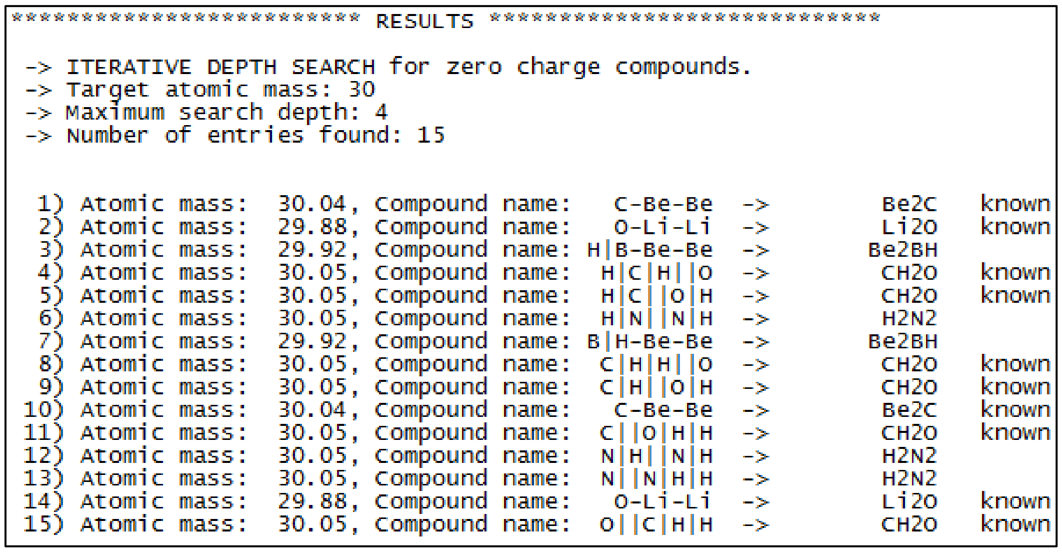
```
************************** RESULTS ******************************

 -> BREADTH FIRST SEARCH for zero charge compounds.
 -> Target atomic mass: 30
 -> Maximum search depth: 4
 -> Number of entries found: 13


  1) Atomic mass:  30.04, Compound name:   C-Be-Be  ->        Be2C    known
  2) Atomic mass:  29.88, Compound name:   O-Li-Li  ->        Li2O    known
  3) Atomic mass:  29.92, Compound name: H|B-Be-Be  ->       Be2BH
  4) Atomic mass:  30.05, Compound name:  H|C|H||O  ->        CH2O    known
  5) Atomic mass:  30.05, Compound name:  H|C||O|H  ->        CH2O    known
  6) Atomic mass:  30.05, Compound name:  H|N||N|H  ->        H2N2
  7) Atomic mass:  29.92, Compound name: B|H-Be-Be  ->       Be2BH
  8) Atomic mass:  30.05, Compound name:  C|H|H||O  ->        CH2O    known
  9) Atomic mass:  30.05, Compound name:  C|H||O|H  ->        CH2O    known
 10) Atomic mass:  30.05, Compound name:  C||O|H|H  ->        CH2O    known
 11) Atomic mass:  30.05, Compound name:  N|H||N|H  ->        H2N2
 12) Atomic mass:  30.05, Compound name:  N||N|H|H  ->        H2N2
 13) Atomic mass:  30.05, Compound name:  O||C|H|H  ->        CH2O    known
```

Depth-first results:



```
************************** RESULTS ******************************

 -> DEPTH FIRST SEARCH for zero charge compounds.
 -> Target atomic mass: 30
 -> Maximum search depth: 4
 -> Number of entries found: 13


  1) Atomic mass:  29.92, Compound name: H|B-Be-Be  ->       Be2BH
  2) Atomic mass:  30.05, Compound name:  H|C|H||O  ->        CH2O    known
  3) Atomic mass:  30.05, Compound name:  H|C||O|H  ->        CH2O    known
  4) Atomic mass:  30.05, Compound name:  H|N||N|H  ->        H2N2
  5) Atomic mass:  29.92, Compound name: B|H-Be-Be  ->       Be2BH
  6) Atomic mass:  30.05, Compound name:  C|H|H||O  ->        CH2O    known
  7) Atomic mass:  30.05, Compound name:  C|H||O|H  ->        CH2O    known
  8) Atomic mass:  30.04, Compound name:   C-Be-Be  ->        Be2C    known
  9) Atomic mass:  30.05, Compound name:  C||O|H|H  ->        CH2O    known
 10) Atomic mass:  30.05, Compound name:  N|H||N|H  ->        H2N2
 11) Atomic mass:  30.05, Compound name:  N||N|H|H  ->        H2N2
 12) Atomic mass:  29.88, Compound name:   O-Li-Li  ->        Li2O    known
 13) Atomic mass:  30.05, Compound name:  O||C|H|H  ->        CH2O    known
```

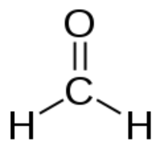
Iterative depth-first results:



```
************************* RESULTS ****************************

 -> ITERATIVE DEPTH SEARCH for zero charge compounds.
 -> Target atomic mass: 30
 -> Maximum search depth: 4
 -> Number of entries found: 15


  1) Atomic mass:  30.04, Compound name:   C-Be-Be  ->       Be2C    known
  2) Atomic mass:  29.88, Compound name:   O-Li-Li  ->       Li2O    known
  3) Atomic mass:  29.92, Compound name: H|B-Be-Be  ->      Be2BH
  4) Atomic mass:  30.05, Compound name:  H|C|H||O  ->       CH2O    known
  5) Atomic mass:  30.05, Compound name:  H|C||O|H  ->       CH2O    known
  6) Atomic mass:  30.05, Compound name:  H|N||N|H  ->       H2N2
  7) Atomic mass:  29.92, Compound name: B|H-Be-Be  ->      Be2BH
  8) Atomic mass:  30.05, Compound name:  C|H|H||O  ->       CH2O    known
  9) Atomic mass:  30.05, Compound name:  C|H||O|H  ->       CH2O    known
 10) Atomic mass:  30.04, Compound name:   C-Be-Be  ->       Be2C    known
 11) Atomic mass:  30.05, Compound name:  C||O|H|H  ->       CH2O    known
 12) Atomic mass:  30.05, Compound name:  N|H||N|H  ->       H2N2
 13) Atomic mass:  30.05, Compound name:  N||N|H|H  ->       H2N2
 14) Atomic mass:  29.88, Compound name:   O-Li-Li  ->       Li2O    known
 15) Atomic mass:  30.05, Compound name:  O||C|H|H  ->       CH2O    known
```

<u>Notable search solutions</u>:

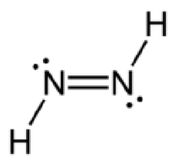
- Formaldehyde (CH_2O): One of the covalent bonding predictions (O||C|H|H) matches its known form, which is shown below:



- Lithium Oxide (Li_2O): The ionic bonding prediction (O-Li-Li) matches its known form. The 1 extra valence electrons in each of the Lithium atoms are ionically shared with the Oxygen atom which lacks 2 valence electrons.

- Beryllium Carbide (Be_2C) : The ionic bonding prediction (C-Be-Be) matches its known form. The 2 extra valence electrons in each of the Beryllium atoms are ionically shared with the Carbon atom which lacks 4 valence electrons.

- Diazene (H_2N_2): Conventionally this compound is written as N_2H_2, which is an exception to the Hill notation for chemical formulas. Because of this inconsistency, the result was not flagged as known. One of the covalent bonding predictions (H|N||N|H) matches its known bond, which is shown below:



Conclusion

In this expert program, every search path was determined by breadth-first, depth-first, or iterative depth-first search strategies. Although heuristics was heavily used to impose restrictions on the generation of child nodes, heuristics was not used to decide the search path. Breadth-first, depth-first, and iterative depth-first search strategies are purely passive search strategies.

SHORTEST PATH THROUGH MAZE

Introduction

A path finding program was implemented to explore the performance of the Dijkstra (a simplified version) and A* search algorithms. Unlike breadth-first, depth-first, and iterative depth-first searches, the Dijkstra and A* searches are active search strategies (i.e. employ heuristics).

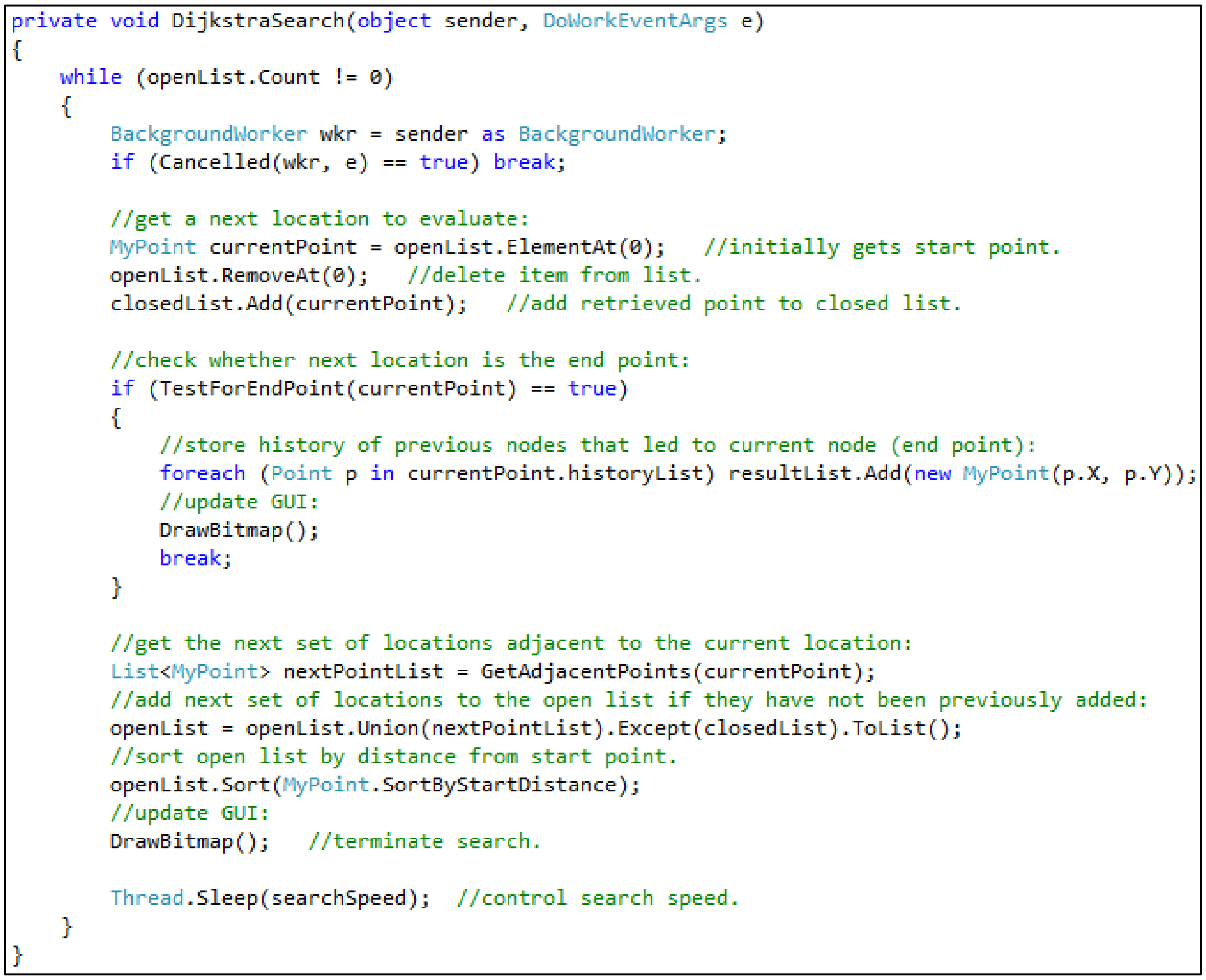
Search Path Strategies

At any current node, possible next nodes (child/candidate nodes) are all the adjacent nodes that are not off-map or coincident with maze walls. Dijkstra and A* search algorithms use different heuristics to select the next node in the search path.

Dijkstra Algorithm

Dijkstra's algorithm selects a next node from a candidate set of possible next nodes based on a candidate node's distance from the search start point. The candidate node with the shortest linear distance from the start point is selected. Thus in an open space search, nodes in all the different directions are explored uniformly in an expanding circular manner. This search strategy is algebraically expressed as:
$f(n) = g(n)$. $g(n)$ measures the unweighted negative distance of any node n to the start node, and $f(n)$ is the fitness function. The shortest distance is less negative and therefore fitter.

The program code to implement a simplified Dijkstra algorithm is shown below.



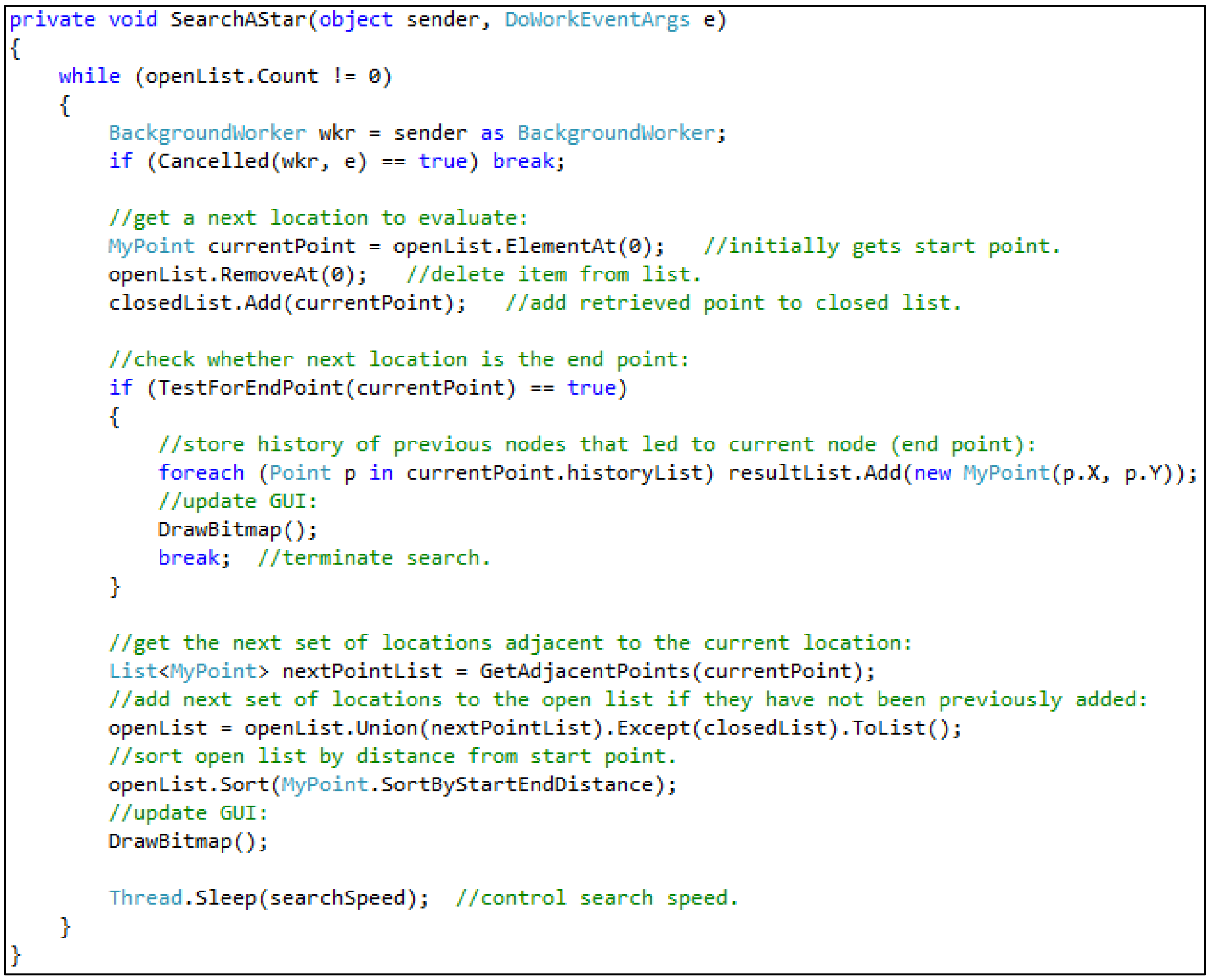
```
private void DijkstraSearch(object sender, DoWorkEventArgs e)
{
    while (openList.Count != 0)
    {
        BackgroundWorker wkr = sender as BackgroundWorker;
        if (Cancelled(wkr, e) == true) break;

        //get a next location to evaluate:
        MyPoint currentPoint = openList.ElementAt(0);   //initially gets start point.
        openList.RemoveAt(0);   //delete item from list.
        closedList.Add(currentPoint);   //add retrieved point to closed list.

        //check whether next location is the end point:
        if (TestForEndPoint(currentPoint) == true)
        {
            //store history of previous nodes that led to current node (end point):
            foreach (Point p in currentPoint.historyList) resultList.Add(new MyPoint(p.X, p.Y));
            //update GUI:
            DrawBitmap();
            break;
        }

        //get the next set of locations adjacent to the current location:
        List<MyPoint> nextPointList = GetAdjacentPoints(currentPoint);
        //add next set of locations to the open list if they have not been previously added:
        openList = openList.Union(nextPointList).Except(closedList).ToList();
        //sort open list by distance from start point.
        openList.Sort(MyPoint.SortByStartDistance);
        //update GUI:
        DrawBitmap();   //terminate search.

        Thread.Sleep(searchSpeed);  //control search speed.
    }
}
```

A* Algorithm

The A* algorithm selects a next node from a candidate set of possible next nodes based on a candidate node's distance from both the search start and end points. The candidate node with the shortest linear distance from the start point plus the *weighted* shortest linear distance from the endpoint is selected. A weight > 1 (3 is ideal) is required in order to keep the search more focused on the endpoint than the start point. This search strategy is algebraically expressed as:

$f(n) = g(n) + h(n)$. $g(n)$ is the unweighted the negative distance of any node n to the start node, $h(n)$ is the weighted negative distance of any node n to the end node, and $f(n)$ is the fitness function.

The program code to implement the A* algorithm is shown below.



```
private void SearchAStar(object sender, DoWorkEventArgs e)
{
    while (openList.Count != 0)
    {
        BackgroundWorker wkr = sender as BackgroundWorker;
        if (Cancelled(wkr, e) == true) break;

        //get a next location to evaluate:
        MyPoint currentPoint = openList.ElementAt(0);   //initially gets start point.
        openList.RemoveAt(0);   //delete item from list.
        closedList.Add(currentPoint);   //add retrieved point to closed list.

        //check whether next location is the end point:
        if (TestForEndPoint(currentPoint) == true)
        {
            //store history of previous nodes that led to current node (end point):
            foreach (Point p in currentPoint.historyList) resultList.Add(new MyPoint(p.X, p.Y));
            //update GUI:
            DrawBitmap();
            break;  //terminate search.
        }

        //get the next set of locations adjacent to the current location:
        List<MyPoint> nextPointList = GetAdjacentPoints(currentPoint);
        //add next set of locations to the open list if they have not been previously added:
        openList = openList.Union(nextPointList).Except(closedList).ToList();
        //sort open list by distance from start point.
        openList.Sort(MyPoint.SortByStartEndDistance);
        //update GUI:
        DrawBitmap();

        Thread.Sleep(searchSpeed);  //control search speed.
    }
}
```

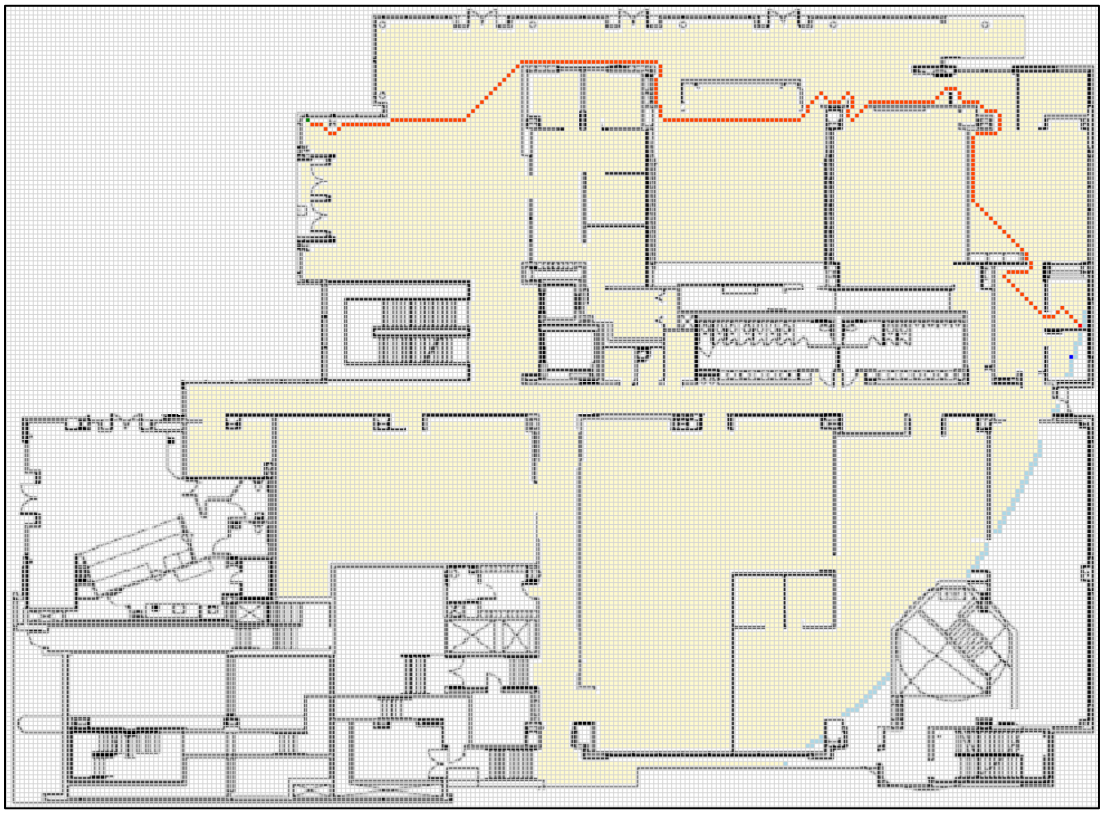
Maze Design

The Maseeh building 1st floor and Cramer Hall 1st floor plans were used to auto-generate mazes. Sealed areas include stairwells, bathrooms, utility areas, and outdoors.

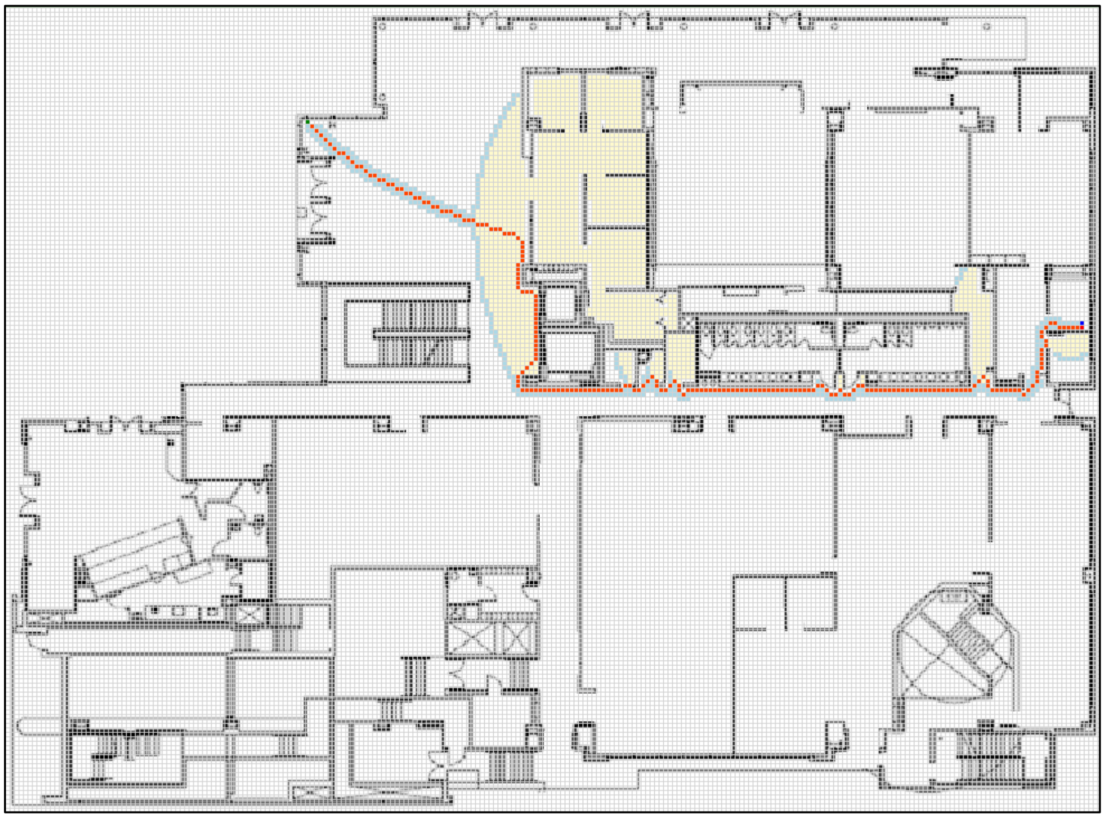
Search Results

Example searches using Dijkstra’s algorithm and the A* algorithm are shown below.

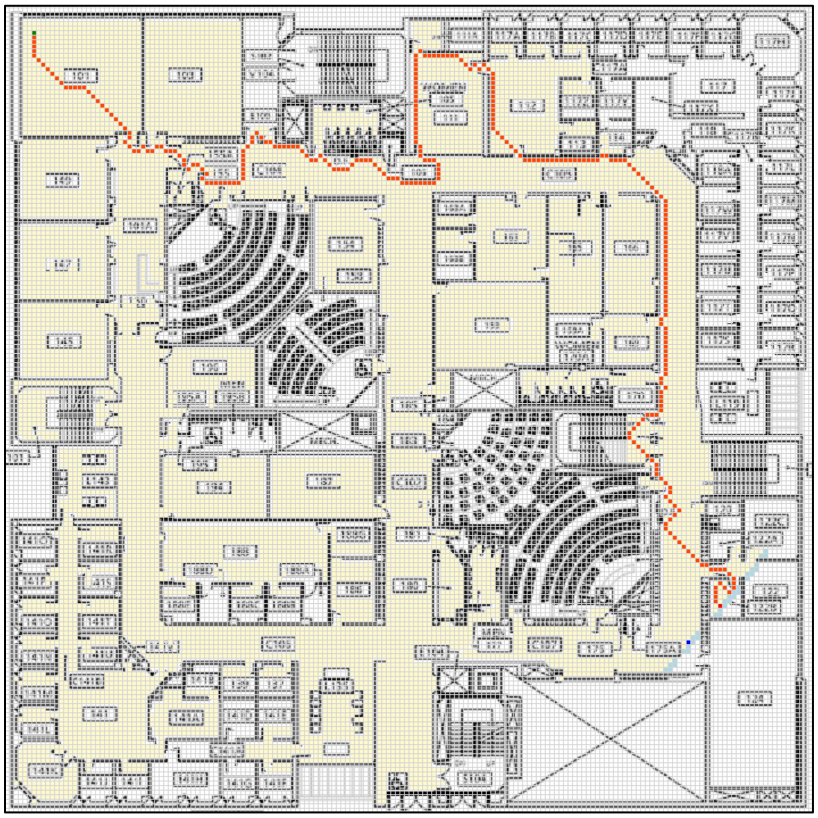
Simulated Maseeh Building 1st Floor, navigated using Dijkstra's algorithm:



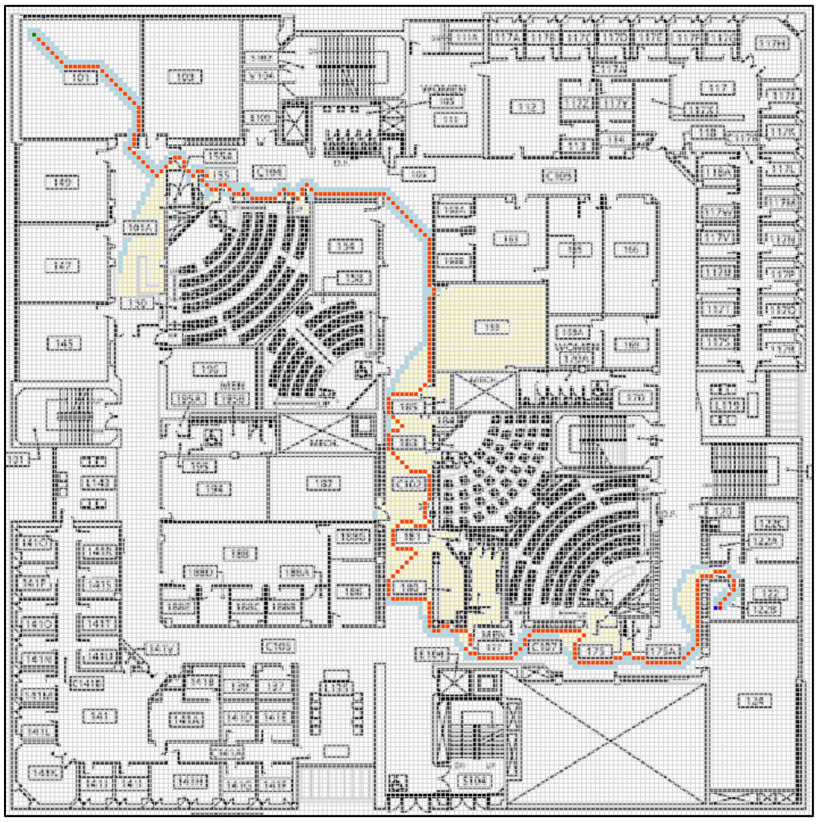
Simulated Maseeh Building 1st Floor, navigated using A* algorithm:



Simulated Cramer Hall 1st Floor, navigated using Dijkstra's algorithm:



Simulated Cramer Hall 1st Floor, navigated using A* algorithm:



Yellow areas indicate previously traversed nodes (closed list). Light blue areas indicate non-traversed potential candidate nodes (open list). The dark orange line indicates the discovered path from start to finish points (result list).

Following is an analysis of the search results for the Maseeh Building 1st Floor search:

Maseeh Building	Total Searchable Nodes (N_S)	Total Traversed Nodes (N_T)	Search Path Node Length	Ratio N_S/N_T
Dijkstra	20228	18514	233	91.5%
A*	20228	2031	220	10.0%

Following is an analysis of the search results for the Cramer Hall 1st Floor search:

Cramer Hall	Total Searchable Nodes (N_S)	Total Traversed Nodes (N_T)	Search Path Node Length	Ratio N_S/N_T
Dijkstra	11947	11135	322	93.2%
A*	11947	1364	311	11.4%

CONCLUSION

Both the A* and Dijkstra algorithms discovered the endpoint, however the A* algorithm did it with less searching and with far more efficiency and speed. The ratio of the actual-searched space vs. total searchable space is a good indication of each algorithm's effectiveness. The A* algorithm is only implementable if the distance to the endpoint can be determined. This extra information allows the A* algorithm to be a more informed and therefore be a superior algorithm.

Simulations with a denser obstacle dense environment (first floor areas filled with obstacles) significantly reduced the performance gap between the two algorithms. This can mainly be attributed to the reduced total traversed nodes for the Dijkstra algorithm.

In this application, every search path was determined by heuristics, except in rare cases where two child nodes had equal suitability. Heuristics had a very limited role in the generation of child nodes, just to disallow child nodes off-map or coincident with maze walls. Dijkstra and A* search strategies are both active search strategies in contrast to breadth-first, depth-first, and iterative depth-first searches.

REFERENCES

A* search algorithm. (n.d.). In *Wikipedia*. Retrieved November 4, 2012, from http://en.wikipedia.org/wiki/A*_search_algorithm

American Chemical Society. (n.d.). STNEasy: Formatting Chemical Formulas. Retrieved November 4, 2010, from http://www.cas.org/training/stneasytips/subinforformula1.html

Braunl, T. (2008). *Embedded Robotics.* Berlin, Heidelberg: Springer-Verlag.

Breadth-first search. (n.d.). In *Wikipedia*. Retrieved November 2, 2012, from http://en.wikipedia.org/wiki/Breadth-first_search

WebQC.Org Chemical Portal. (n.d.). *Calculate molecular weight - molar mass calculator*. Retrieved November 1, 2012, from http://www.webqc.org/mmcalc.php

Depth-first search. (n.d.). In *Wikipedia*. Retrieved November 2, 2012, from http://en.wikipedia.org/wiki/Depth-first_search

Dictionary of chemical formulas. (n.d.). In *Wikipedia*. Retrieved November 3, 2012, from http://en.wikipedia.org/wiki/Dictionary_of_chemical_formulas

Dijkstra's algorithm. (n.d.). In *Wikipedia*. Retrieved November 4, 2012, from http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Iterative deepening depth-first search. (n.d.). In *Wikipedia*. Retrieved November 4, 2012, from http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

Luger, G. (2009). *Artificial Intelligence.* Boston, MA: Pearson.

Portland State University. (2010). *First Floor Plan Engineering Building*. Retrieved November 5, 2012, from http://www.fap.pdx.edu/floorplans/docs/EB_f1317916752.pdf