**CS 5600 - Artificial Intelligence**

Report

Project 1: Search

Christopher H. Weber  
 9/27/2009

**Table of Contents  
  
1. Preliminaries  
 1. A Boilerplate Code Changes   
 1. B How To Run The Program  
  
2. Program Design  
 2.A Node Design  
 2.B Agent Function  
 2.C Tree and Graph Searches  
 2.D Expand and Successor Function  
 2.E Miscellany  
  
3F. Figures - Successful Searches per World Size  
3. Results  
 3.A Successful Searches per World Size  
 3.A.1 How Successful Searches Drive the Remaining Results Analysis  
 3.B Uninformed Search Performance on the 8-Grid with Duplicate Detection  
 3.B.1 Average Total Number of Nodes Expanded  
 3.B.2 Average Total Time of the Search  
 3.B.3 Average Total Score  
 3.B.4 Average per Node Cost  
 3.B.5 End  
 3.C Informed Search Performance Across All Grids with Duplicate Detection  
 3.C.1 Average Total Number of Nodes Expanded (across all grids)  
 3.C.2 Average Total Time of the Search (across all grids)  
 3.C.3 Average Total Score (across all grids)  
 3.C.4 Average per Node Cost (across all grids)  
 3.C.5 End  
  
4. Results and Search Algorithms Interpreted  
 4.A Reviews of Each Search Over Its Variations  
 4.A.1 Uninformed Searches and Tradeoff Comparisons  
 4.A.1.a Depth First Search  
 4.A.1.b Breadth First Search  
 4.A.1.c Uniform Cost Search  
 4.A.1.d Iterative Deepening Depth First Search**

**4.A.2 Informed Searches and Comparison of Tradeoffs  
 4.A.2.a Best First Search  
 4.A.2.a A\* Search  
 4.B Heuristics and their Effect on Informed Searches  
 4.B.1 Straight Line  
 4.B.2 Manhattan  
 4.B.3 Homegrown  
 4.B.3.a Homegrown Heuristic Explanation  
 4.C Total Time and per Node Cost Analysis  
 4.D Cycle Checking vs. Duplicate Detection  
 4.E Uninformed Search vs. Informed Search  
  
  
Appendices  
 Informed Search Performance Charts by Grid Size  
 AI.1 - raw data - 40 tests  
 Source folder for AI.1**

**1. Preliminaries  
  
1. A Boilerplate Code Changes**

The Simulation and WorldApplication classes are changed for output purposes only.

In the WA class there now exists a static outputWriter accessible to all searches. If there is no declaration of this object in the WA class, no search will run. This is for data generation to a file.

The file output looks like: <size>:<seed>:<nodeCount>:<time(ms)>:finalScore.

In both the Simulation and WA class, I have altered the screen output. Instead of a step by step iteration of the changing grid, the screen shows:

introduction information;  
initial grid;   
first step marker and score;   
and the results of the search.

If results of the search are generated (memory heap errors can occur), the results are shown as:

is the goal found;  
 final counter value;  
 total time;  
 solution action sequence (in ints);  
 and the final score.

**1. B How To Run The Program**

In the AgentFunction class there is a method called search. In order to run any search, one must uncomment the desired search before running the program.   
   
 There is a need to have many steps available to the WW program due to the inefficiency of the depth first search solutions. Thus, in addition to the command line arguments given, one must also expand the max steps value. The resulting command line arguments are:

-r <seed> -a false –d <dimension> -u true -s 100000

where -s is the number of steps allowed in the trial.

Simply load the src folder provided into an IDE to begin…

**2. Program Design**

**2.A Node Design**

The most vital aspect of the Node class is a static reference to an instance of GridState. This reference contains the initial state’s 3D char array amongst other data. The Node class has an initial node constructor accepting this GridState object.  
 The Node class then uses data members of int action, XYPair location, char direction, a XYPair list of killed Wumpii, and a boolean hasGold to record valid state changes to this static initial GridState object. This reduces the size of any Node object from 5 Kb large to less than 100 bytes.   
 Of course the Node class also contains the data members of: a parent node reference; depth; stepCost; g\_value (current total path cost); h\_value (a heuristic value); and f\_value (an advanced heuristic value).   
 An initial Node constructor for the informed searches accepts an int typeHeuristic and an XYPair goldLocation. This goldLocation object is obtained from a Problem class instance, which also contains a Goal object.  
 Subsequent node creation uses a constructor which accepts the new action assigned to it by the class ExpandAndSuccessorFunction. This constructor contains a switch based on the heuristic accepted by the initial Node constructor to update the f and h values of the node instance.  
 The Node class also implements a method, getPathFromRoot, which yields the path from any node to the root of the tree or graph.  
 The Node class also overrides the java Object class equals and hashCode methods. This enables a hash set for the closed list, which produces an O(1) lookup time.

**2.B Agent Function**

The AgentFunction class is an orthodox implementation of the Simple Problem-Solving Agent as outlined in the course textbook. Further, it uses as its basis the Java code provided by the author. If one were to compare the code of the AgentFunction to the book’s pseudocode algorithm, one would find a line by line correspondence.   
 The process method uses the initial TranferPercept instance to create Goal and Problem objects. The Problem object is sent to the AgentFunction search method. If search finds no solution, the process method returns NO\_OP actions for the duration of the trial.  
 The AgentFunction search method contains all search variations as outlined in the project specifications. One must un-comment the desired specific search in order to run the program.

Search receives the goalNode returned by the selected inner search function. It calls the Node class’s getPathFromRootMethod. It takes the action contained in each node along the path and loads the actionSequence list expected by AgentFunction’s process method.  
 Goal contains the XYPair locationOfGold and a Boolean method isGoal. Problem contains the initial GridState object and the Goal instance.

**2.C Tree and Graph Searches**

Just like the AgentFunction class, all Search classes are orthodox implementations of the book’s pseudocode algorithms. If one were to compare the project’s code for tree and graph searches to the book’s pseudocode algorithms, one would find an exacting correspondence.  
 Depth First Search and Iterative Deepening Depth First Search implement LIFO queues. Breadth First Search uses a FIFO queue. Uniform Cost Search, Best First Search, and A\* Search use priority queues.  
 Each node is evaluated using the goal class method isGoal. A Goal instance was accepted as part of the Problem instance received by the search class’s constructor. If a graph is used for the search, then a hash set of type Node is implemented. There are timers and print statements to the display and to a file.   
 A key component of all these searches is the call to method expand of the ExpandAndSuccessorFunction class.

NOTE: Though all seeds for all size worlds are solvable, every tree/graph search is prepared to return whether or not the world has a solution.

**2.D Expand and Successor Function**

Too, this class is an orthodox implementation of the book’s pseudocode algorithms. The method expand builds the list of successors based on the possible actions available to the agent. As discussed in a professor’s note, it is the successor method which evaluates whether a successor node is valid. Validity simply means the successor node’s action changes the state of the world. If the action sent by the expand method produces a valid state change, a new node is returned with that new state.

NOTE: It is possible to trace the ExpandAndSuccessorFunction’s process by selecting “true” for its variable debug.

**2.E Miscellany**

The XYPair class contains an x,y pair. It overrides the Java Object class methods equals and hashCode, and implements the interface Comparable.  
 The Comparator classes of UCS, BestS, and AStarS allow Java Priority Queues to compare Nodes over the values of g (path cost), h (distance) , and f (g + h).  
 IDDFS with duplicate detection implements a global closed list. This means that for each iteration, only one level of depth exists in the fringe. Of course, as with all searches implementing duplicate detection, this means all nodes live on in the closed list.  
 A recursive version of the IDDFS without enhancements is implemented. This version runs to 4 levels greater depth than the purely iterative raw IDDFS. This is a tremendous performance increase that must be explained by the nature of using a stack (recursive method call) to handle the life of valid successor nodes.  
 The print results portion of each search does obscure the important code a bit. Given more time, this is a flaw which would be corrected by implementing that same code as a method.

Successful Searches per World Size

Grid Size 4

Grid Size 8

Grid Size 16

Grid Size 32

**3. Results** NOTES: The remainder of this report uses initials for the various searches and heuristics: DFS, BFS, IDDFS, UCS, Best, A\*, and H, M, and S.  
 The grid size is indicated by the notation <size>-grid.

**3.A Successful Searches per World Size**

The most revealing statistics for general search performance are shown in these tables.  
 Raw searches (no cycle-checking or duplicate detection) produced spotty performance in general. Only BFS and IDDFS were able to produce complete results for the 4-grid. This is because of their 1:1 step-to-depth operation.  
 Cycle-checking was able to improve search success in a few cases on the 4-grid. A cycle-check is defined thusly: a node must not be equal to its parent or grandparent node. Duplicate detection was the key to performance for all search algorithms. Duplicate detection is defined as a closed list. A closed list is implemented as Java Hash Set (with O(1) lookup performance) and eliminates all repeating states (equal nodes).   
 Cycle-checking and duplicate detection were largely redundant, as a closed list subsumes the cycle check’s purpose.  
 Further details over these statistics will be discussed in the Results Interpreted section.

**3.A.1 How Successful Searches Drive the Remaining Results Analysis**

All uninformed searches using duplicate detection were able to solve the 8-grid. Thus, this grid will be used to compare uninformed search performance in section 3.B.  
 All informed searches with duplicate-detection were able to solve all size worlds. Thus, their statistics will be aggregated into tables comparing their performance in section 3.C.

The following tables will show these components of the test data:

* Total Number of Nodes expanded (a primary key to a successful search).
* Total Search Time (a direct relationship to the total number of nodes expanded).
* Final Search Score (reflecting the nature and function of the algorithm design).
* per Node Cost (the number of nodes expanded over the total time)

NOTE: Expanded node count comparisons over raw search, cycle-checking, and duplicate detection are possibly interesting. However, the total successful searches is the most simple indicator of performance. Combining this data with the following data in this section yields comprehensive results. Therefore, discussion of reduction of expanded node count totals over these enhancements of raw search algorithms is reserved for the Results Interpreted section.

**3.B Uninformed Search Performance on the 8-Grid with Duplicate Detection**

As previously stated, all uninformed searches solved all seeds for the 8-grid only if using duplicate detection. This was the maximum size grid all searches were able to solve without a JVM memory heap error. Thus, the 8-grid statistics are shown in these tables. Results are averages over the 3 seeds.

**3.B.1 Average Total Number of Nodes Expanded**

DFS creates the minimum number of expanded nodes. UCS creates more nodes but still relatively minimal. IDDFS improves over BFS’s maximum expanded node creation.

**3.B.2 Average Total Time of the Search**

As suggested in the previous section, time is directly correlated with the total number of expanded nodes.

**3.B.3 Average Total Score**

|  |  |
| --- | --- |
| Type | **Average Score** |
| **BFS** | 966.33 |
| **DFS** | 642.67 |
| **UCS** | 976.67 |
| **IDDFS** | 972.33 |

Because the DFS score is so low, a normal graph renders the other searches values as equivalent. UCS’s score is in fact the maximum possible score. BFS and IDDFS are able to approach this score, while DFS lags far behind. The BFS score is less than that of the IDDFS.

**3.B.4 Average per Node Cost**

All per node costs hover at about .04 ms per node expanded.

**3.B.5 End**

These results will be discussed in the section Results Interpreted.

**3.C Informed Search Performance Across All Grids with Duplicate Detection**

As previously stated, all informed searches using duplicate detection solved all seeds for all grids. Because their performance is uniform across the grid sizes, and to prevent chart overload, results for all grid sizes are aggregated into tables to show their performance. Thus, results shown are averages over the 3 seeds for all grid sizes.

NOTE: If one desires, one may review the additional file which shows performance over each grid size.

**3.C.1 Average Total Number of Nodes Expanded (across all grids)**

Of the A\* search variations, the H heuristic creates the least number of nodes, and the S heuristic the maximum. The Best variations are nearly equal.

**3.C.2 Average Total Time of the Search (across all grids)**

As suggested in the previous section, time is directly correlated with the total number of expanded nodes.

**3.C.3 Average Total Score (across all grids)**

The A\* search heuristic variations appear to be equal. The H heuristic outperforms the other Best heuristic variations.

**3.C.4 Average per Node Cost (across all grids)**

**3.C.5 End**

These results will be discussed in the section Results Interpreted.

**4. Results and Search Algorithms Interpreted** Due to the number of tests run the JVM’s heap size was left at its default size of 128MB. Given that some searches performed for all size worlds on all seeds, it can be argued that this heap size restriction gives sufficient results by which to compare the performance and features of all search algorithms.   
 Because no searches were able to complete all seeds for even a 4-grid in their raw or cycle-check variations, no valid averages could be generated for analysis (How does one average infinity?). However, the Successful Searches per World Size table yields data enough to interpret each search’s raw and cycle-check variations. More concrete data is available in the accompanying raw data Excel file. Occasional references to results not provided in this main report body exist in that file, which is organized to the extent that such data is easily found.  
 Time totals are not discussed as much as expanded node count, as the two metrics directly correspond, with total count being the primary determiner of total time. However, a section below does review this assertion.

**4.A Reviews of Each Search Over Its Variations**  
 **4.A.1 Uninformed Searches and Tradeoff Comparisons**

Reviewed throughout the following subsections.

**4.A.1.a Depth First Search**

DFS in its raw form drives left down the tree, as it is allowed to repeat states endlessly. The 2-deep cycle check is not enough to halt this behavior, as every change in direction is a valid state change, and the repetition of state is 4 levels higher than the current state. However, with duplicate detection DFS becomes the most effective of uninformed searches at finding the gold, finding gold for all seeds for all world sizes. By the nature of its construction, a LIFO queue, it is spare in its node expansion but yields inefficient paths. For instance, due to the ordering of actions in the Expand method, DFS will always take 3 left turns rather than one right. This means poor final scores result compared to all other searches.

**4.A.1.b Breadth First Search**

BFS in its raw form is able to solve all 4-grid seeds, a quality shared only with IDDFS. Cycle checking does reduce the number of nodes expanded, but not enough to affect finding solutions of these seeds. With duplicate detection it is able to solve all 8-grid seeds. By the nature of its construction, a FIFO queue, it performs a level-by-level expansion. Thus, it expands more nodes than any other search. It is restricted in the level of depth to which it can search, generally stopping at 13 levels of depth (given the JVM heap size). Therefore, any grid whose gold lies beyond 13 steps is outside the space available to BFS. However, it does a fine job on small search spaces.

**4.A.1.c Uniform Cost Search**

UCS is able to solve very little without the benefit of duplicate detection. However, in solving a seed in the 4-grid where its raw form fails, it does show how the limited 2-deep cycle check has a measurable effect on node expansion. By the nature of its construction, a priority queue, given that successor nodes incrementing their costs by 1 are also repeating states, too many nodes are generated to find an eventual solution that has a minimum cost of, say 11. However, with duplicate detection, UCS provides its intended maximum possible scores, while delivering a minimum possible node expansion. Why it largely fails over the 32-grid is the abundance of equal cost possible successor nodes, overwhelming the heap with valid node objects in the closed list.

**4.A.1.d Iterative Deepening Depth First Search**

IDDFS is the strongest performer of all uniformed searches in its raw and cycle-checking forms. By the nature of its construction, a LIFO queue with a depth limiter, the nodes on cutoff paths are allowed to die from the heap as they are popped off the fringe.   
 A key feature of IDDFS is that even if the node’s depth has reached the cutoff, in being popped off the fringe a goal test can be performed. Though that node is not expanded, it is still evaluated. Thus, IDDFS saves an entire level of expansion of the search space over BFS. More, the order in which nodes are popped is reversed from BFS, yielding a possible different goal node. This is the reason why the IDDFS score average is different from BFS.   
 Of course, duplicate detection does enable IDDFS to achieve deeper depth levels, as the cycle-checking protocol as implemented is unable to prevent the occurrence of most common cycles. However, this duplicate detection prevented the natural death of nodes as paths ended in the cutoff.   
 Expanded node count is less than only the BFS, but its total score is second only to UCS.

**4.A.2 Informed Searches and Comparison of Tradeoffs**

The heuristics mentioned here will be reviewed in a later section.

**4.A.2.a Best First Search**

Best expands few nodes, takes little time, and gives sub-optimal paths to the goal. By the nature of its construction, a priority queue, it uses a heuristic to drive its node expansion choices. This heuristic gives a value stored in each node as h\_value. The nature of that heuristic lightly affects expanded node count. The heuristics used for Best are Manhattan, Straight Line, and Homegrown. The Homegrown heuristic gave best average score, but expanded slightly more nodes.

**4.A.2.a A\* Search**

A\* expands far more nodes than Best. However, it returns optimal scores. By the nature of its construction, a priority queue, in addition to the heuristic used by best (and recorded in the node as h\_value), it uses the total path cost thus far for the current node (the same used by UCS). These two values aggregate into the node data member f\_value. The Homegrown heuristic created the least amount of nodes, but was found to be inadmissible, scoring 2 points below optimum on one seed for the 4-grid. The Straight Line heuristic generates the most nodes.

**4.B Heuristics and their Effect on Informed Searches**  
**4.B.1 Straight Line**

S calculates the distance to goal “as the crow flies”, the Pythagorean Theorem’s hypotenuse. For A\* it expands the most nodes by a large margin, while for Best node expansion it is roughly equal to the other heuristics. For A\* it provides an optimal score, and for Best a greater score than M. The reason why H is a less efficient heuristic than M is because its estimate of total cost is far more conservative than could possibly be achieved.

**4.B.2 Manhattan**

M calculates the distance to goal by counting the right triangle’s “a” and “b” sides, i.e., the number of cells to goal by walking the shortest straight line(s) possible (allowing up to one change of direction). For A\* it provides an optimal score, and for Best the least score. For both Best and A\* it generates the nearly the least number of expanded nodes.

**4.B.3 Homegrown**

H creates the least amount of nodes for A\* and the most for Best. The A\* search tests reveal H to be inadmissible, scoring 2 points below optimum on one seed for the 4-grid. However, it yields far higher scores for Best. As Best does not account for current path cost, the H heuristic informs Best of the cost of shooting arrows, whereas M and S do not account for the cost of this action.

**4.B.3.a Homegrown Heuristic Explanation**  
 H incorporates M as its base, thus M’s fine performance is acquired for further tuning. After M’s calculation of f\_value, if certain conditions are met by the node, H adds a few points to the node’s h and f values. These conditions are:

* If a node was facing a pit or wumpus and for its current action turns away, then add 1 to the h\_value (and thus the f\_value, as M is H’s base) of the node, unless the pit(P) or wumpus(W) lies next to a world edge.
* If a node action is “shoot arrow”, then add 5 to the h\_value (and thus the f\_value, as M is H’s base).

Consider the minimal case where an agent is on a straight line to the gold, but a P/W blocks its path. Use the Manhattan method for calculating distances.

* The current M cost is 1 less than the agent being in a cell that is one step removed from being in a straight line to the gold.  Consequently, add the M cost of being 1 cell further away than optimal, for that is the action the agent is about to take. Though the next node will also consider that fact, the cost of moving around a W/P back to the straight line path is greater than 1, so this is a conservative addition.
* The cost of killing a W is 10. Optimally, the agent wants to go around the W back to the straight line path to the gold. The Manhattan cost plus the cost of the W kill is 11 (if the W was 1 step away from the G). The Manhattan cost of moving around the W to the G is 6  (3 from, 2 from, 1 from). 1 will have already been added from the previous case, thus the total added M-based cost is 7. The difference between the two costs is 4. Make the cost of killing a W greater than the difference between the two possible action sequences to discourage the unnecessary killing of Wumpii.

This heuristic is inadmissible because in fact, for this minimal case, the cost of killing a W is 12, not 11. The steps to the gold are 2, not 1. Thus, the search is able to make a suboptimal decision.

**4.C Total Time and per Node Cost Analysis**

As previously mentioned, time is directly correlated with the total number of expanded nodes. Time measurement is subject to the vagaries of the operating system and Java Virtual Machine. But let the following examination of the informed searches over total time and average per node cost clarify this contention.  
 The Best searches appear to take more time per node to complete. However, the total number of expanded nodes in the Best searches is quite low. Thus, the greater time per node may simply have been a vagary of the JVM.   
 It is more reliable to state how many operations each search/variation/heuristic performs. For instance, the H heuristic calls the M heuristic and then performs many more operations. Also, the S heuristic calls Java Math class routines. These variations on their own may be isolated and tested for true cost. One must realize that A\* and Best are identical save that Best does not utilize the pre-computed current total path cost.   
 Given that the number of tests per seed was not an average over many trials, and that the JVM was given to return time in milliseconds incremented in values of 15 or 16, one cannot safely conclude by these statistics that perNode time cost for the Best searches are definitively higher than A\*’s. To gain a more accurate assessment of total time beyond that of total expanded node counts, one would have to perform more trials over the same seeds, search type and variation.

**4.D Cycle Checking vs. Duplicate Detection**

Cycle checking is potentially far more powerful thanimplemented in this report. One might implement several additional checks: do not allow 3 lefts instead of a single right; check the 4 level distant node for repetition of state; mark nodes as traversed; all these are valuable enhancements to the current cycle checking protocol.  
 Eliminating a closed list allows nodes to die if they are not a parent of a valid node on the fringe. DFS and IDDFS in particular would be greatly enhanced by a powerful cycle checking solution. DFS’s final score would be greatly improved. IDDFS would not need to save all expanded nodes, which defeats one of its primary intended qualities. Duplicate detection prevents the natural death of nodes as their paths end in the cutoff and the search moves up the tree and to the right to examine other paths.  
 However, duplicate detection alone serves extremely well for all searches. It is the single key enhancement on the searches which enables them to perform to completion, often with surprisingly low expanded node counts.  
 Lastly, the combination of both cycle-checking and duplicate detection is utterly redundant, as there is no change in expanded node count from that of duplicate detection alone.

**4.E Uninformed Search vs. Informed Search**

If one does not know where the goal lies in the search space, it is harder to develop a heuristic. Further, it is conceivable that there are some problems for which no heuristic has been developed. DFS enhanced with duplicate detection seems an ideal choice for such situations, as is has great economy of node expansion and solves all seeds for all grid sizes. Too, UCS does well evaluating least cost paths. Certainly, if the goal location is not known, these searches can be used carefully to find a goal.  
 However, if the goal state location is known and a heuristic can be developed to find it, there is no advantage to using uninformed search. All informed searches, when implementing a closed list, solve all grids for all seeds. They do so with a far higher score and a lower total node count compared to DFS, the most powerful of the uniformed searches. (One may check the 8-grid results of section 3.B against the Informed Search Performance Charts by Grid Size document included as a separate appendix to this report.)   
  
  
  
**Appendices  
 Informed Search Performance Charts by Grid Size  
 AI.1 - raw data - 40 tests  
 Source folder for AI.1**