10. Single-agent Search

Jonathan Schaeffer
jonathan@cs.ualberta.ca
www.cs.ualberta.ca/~jonathan

Moving On…

- Two-player adversary search is nice, but not all interesting problems can be mapped to games
- Large class of optimization problems that all have the same search properties
- Find the best search value from the perspective of a single player
- Single-agent search

Applications

- Pathfinding
- Dynamic programming
- Job shop scheduling
- DNA sequence alignment
- Scheduling
- Planning
- Constraint satisfaction
- …

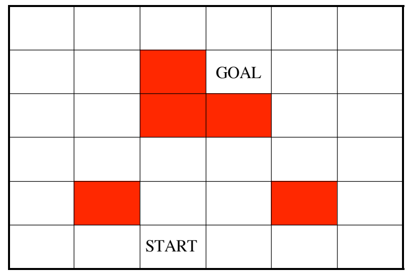
Why Alpha-Beta First?

- Many of the performance enhancements we saw in alpha-beta translate to single-agent search
- Most originated with alpha-beta, and were adopted by other classes of search algorithms

Application: Pathfinding

- Consider a sample application
- Find a *minimal cost* path from a start node to a goal node
- Can move one square horizontally or vertically, each with a cost of one
  - Can be generalized to include diagonals
  - Can be generalized to include variable costs

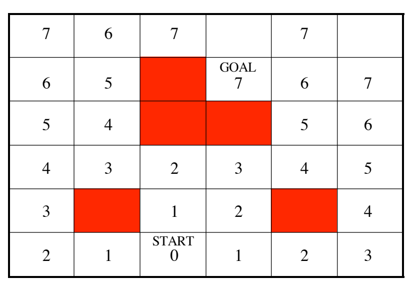
Application



Solution 1

- Trivial solution
- Explore outward from the start node until reaching the goal node
- Can use iterative deepening to guarantee minimal cost path
  - Try paths of length 1, then 2, etc.

Solution 1



Solution 1

- Note that more than one path can lead to a node
  - Some of these paths are non-optimal
- Note that cycles are possible
- Observation: we need to eliminate duplicate states!

9/9/02 9

Solution 2

- Trivial observation that searching to depth 1 is a waste of time since we are obviously more than 1 away from the goal
- Add to the search an evaluation function that estimates the distance to the goal
- What is a simple estimator of distance?

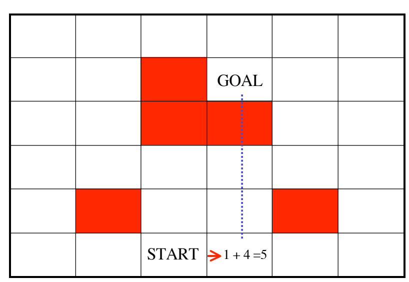
9/9/02 10

Solution 2

- For pathfinding, a good *estimate* of distance to go is the Manhattan distance
  - Number of horizontal and vertical moves to the goal node
- Cost of reaching a node is now two parts:
  - Distance already traveled
  - Estimate of distance to go
- If the cost of a node exceeds the iterative deepening threshold, then stop searching that path

9/9/02 11

Manhattan Distance



9/9/02 12

Pathfinding

	5 + 2 = 7		GOAL 7 + 0 = 7	6 + 1 = 7	
	4 + 3 = 7			5 + 2 = 7	
	3 + 4 = 7	2 + 3 = 5	3 + 2 = 5	4 + 3 = 7	
		1 + 4 = 5	2 + 3 = 5		
	1 + 6 = 7	START 0 + 5 = 5	1 + 4 = 5	2 + 5 = 7	

IDA*

- Iterative deepening A*
- The cost of a node is (using A* terms)
  - f = g + h
  - g = cost incurred to get to this node
  - h = heuristic estimate of getting to goal
- Iterative deepening iterates on a threshold
  - Search a node as long as f <= threshold
  - Either find a solution (done), or fail, in which case the threshold is increased and a new search started

IDA* (1)

```
threshold = Eval( s );
done = false;
while( not done ) {
   done = IDA*( s, 0, threshold );
   if( done == false ) threshold++;
}
```


IDA* (2)

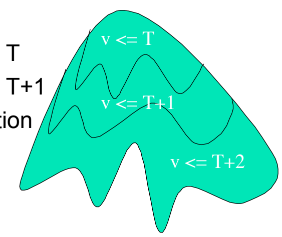
```
IDA*( state s, int g, threshold t ) {
   h = Eval( s );
   if( h == 0 ) return( true );
   f = g + h;
   if( f > threshold ) return( false );
   for( i = 1; i <= numchildren; i++ ) {
        done = IDA*( s.child[ i ], g + cost( child[ i ] ), t );
        if( done == true ) return( true );
   }
   return( false );
}
```

IDA* Comments

- Automatically builds a variable-depth search
  - Provably bad lines are cutoff as soon as possible
  - When the cutoff occurs depends on the quality of the evaluation function
- Storage requirements are trivial; just the recursion stack
- Iteration *i+1* repeats all the work of iteration *i*!
- For some domains you can do better than iterate by 1
  - Use the mimimum f-value seen at a leaf node during an iteration as the next threshold

IDA* Tree

- Depth-first search
- Root's value = T
- Search nodes <= T
- Search nodes <= T+1
- Repeat until solution



IDA* Comments

- Is IDA* guaranteed to produce an optimal answer?
- Yes!
- But only if…
- The evaluation function has to be *admissible*:
  - It must always be a lower bound on the true solution length

Manhattan Distance

- Computes a direct path from a node to the goal
- Ignores all obstacles, which can only lengthen the path
- Therefore it is an admissible heuristic

Monotonicity

- Most admissible heuristics also have the monotonicity property
- The f values never decrease along a path if monotonicity holds
- If you have a non-monotonic heuristic, one can always modify the search to make the heuristic monotonic…
  - How?

Examining h

- Simplified cost of a search
- Uniform branching factor b
- Search depth d
- Ignore all other enhancements
- No heuristic: b^d
- Average heuristic value is h: b^{d-h}
- The quality of the heuristic has an enormous impact on the search efficiency

Examining h

- What does it mean to iterate?
- If the first iteration finds an answer, then h had no error
- If a second iteration is required, then there is an error of 1 in h
- The number of iterations indicates the degree of error in h

Eliminating Redundant Nodes

- Need to eliminate duplicate nodes
- Trivial optimization for many domains is to disallow move reversals
- For more sophisticated detection of redundant nodes, we can use a transposition table

Transposition Table

- Store the t and g values in the table, and only search a transpositon node with the smallest g, and only once for the current t
- Use table only to indicate which nodes *not* to search
- No need to store values, since the search stops when a solution is found
- All other TT issues (table size, hashing, table entry replacement) remain the same as for two-player games

Sliding Tile Puzzle

Sam Lloyd’s creation was the Rubik’s Cube of the 1800s.

Experiments

- Korf problem set of 100 positions

- Search 36700
- Search - move reversals 100
- Search + TT (256K) 37

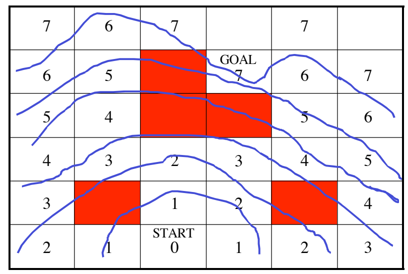
A*

- Single-agent search began in the 1960s with the A* algorithm [2]
- This algorithm dominated AI search for two decades, but has competition now from IDA*
- Why teach IDA* first? Easy to explain once you’ve seen Alpha-Beta

A*

- Each iteration of IDA* re-searches the tree over again beginning at the root
- All that overhead can be eliminated…
- … by keeping track of the *search frontier*, and only expanding nodes on the frontier
- A* is a *best-first* search algorithm

Search Frontier



A* Data Structure

- OpenList
  - List of nodes in the tree that are not yet fully considered
  - Ordered from best to worst *f* value
- ClosedList
  - Nodes that have been fully expanded
  - No longer on any optimal path

A* Algorithm (1)

- Take best (first) node from OpenList
  - Check for solution
  - Expand all the children
  - Move node to the ClosedList
  - As far as we know, done with this node

A* Algorithm (2)

- Expanding a child
  - Check if seen before Open/ClosedList
    - If the node has been seen before with the same or better g value, then reject
  - Add to OpenList for consideration
- In effect the lists act as a cache of previously seen results
- NOTE: the algorithm requires all nodes to be in these lists, unlike a TT

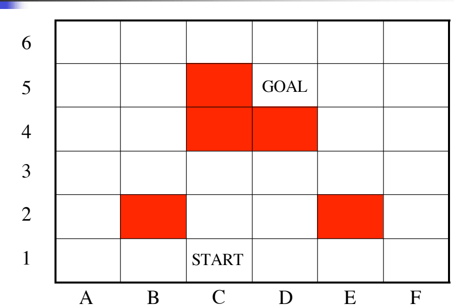
A* (1)

```
A*( state s ) {
    s.g = 0; s.h = Eval( s ); s.f = s.g + s.h; s.parent = null;
    done = false;
    push s on OpenList
    while( OpenList != empty && done == false ) {
        pop s from head of OpenList
        if( s is a goal node ) { done = true; break; }
        foreach( i = i; i <= Children( s ); i++ ) {
            Consider( s, s.child[i ] );
        }
        add s to ClosedList
    }
    return( done );
}
```


A* (2)

```
Consider( state from, state to ) {
    newg = from.g + Cost( from, to );
    if( ( to is in OpenList or ClosedList ) and
        ( to.g <= newg ) )      return;
    to.g = newg; to.h = Eval( to );
    to.f = to.g + to.h; to.parent = from;
    if( to is in ClosedList ) remove to from ClosedList
    if( to is not in OpenList ) insert to in OpenList sorted
                                              by f-value
}
```


Example



Example

- Step 1: Initialize
  - (C1, 0 + 5 = 5, null)
  - ( )
- Step 2: Expand C1
  - (C2, 1 + 4 = 5, C1) (D1, 1 + 4 = 5, C1) (B1, 1 + 6 = 7, C1)
  - (C1, 0 + 5 = 5, null)

Example

- Step 3: Expand C2
  - (C3, 2 + 3 = 5, C2) (D1, 1 + 4 = 5, C1) (D2, 2 + 3 = 5, C2) (B1, 1 + 6 = 7, C1)
  - (C1, 0 + 5 = 5, null) (C2, 1 + 4 = 5, C1)
  - Why isn't C1 added to the OpenList?
  - C1 is found in the ClosedList with a lower g value

Example

- Step 4: Expand C3
  - (D3, 3 + 2 = 5, C3) (D1, 1 + 4 = 5, C1) (D2, 2 + 3 = 5, C2) (B1, 1 + 6 = 7, C1) (B3, 3 + 4 = 7, C3)
  - (C1, 0 + 5 = 5, null) (C2, 1 + 4 = 5, C1) (C3, 2 + 3 = 5, C2)

Sorting Open List

- Sort by increasing *f* value, but what about ties?
- Break ties based on *g* value
  - Larger g values mean more accurate information and less heuristic approximation

A*

- Does not have the iterative overhead of IDA*
- Only expands nodes that are shown to be relevant
- Needs to maintain a history of all nodes previously searched
- In practice, faster than IDA*, but A* runs out of memory very quickly!

9/9/02 41

IDA* versus A*

- For many types of problems, IDA* flounders in the cost of the re-searches, causing many to prefer A* over IDA*
  - Why?
- But… IDA* is handicapped with no storage!
  - A* uses a closed list -- in effect a perfect cache of previously seen states
  - IDA* uses almost no storage
  - IDA* with a transposition table can be competitive with A*

9/9/02 42

Which to Choose?

- IDA* is guaranteed to work, albeit possibly more slowly
- A* is more efficient, but can run out of memory
  - Can also run slower because of cache effects
- The right choice depends on properties of your application

9/9/02 43

References

[1] R. Korf. “Best-first Iterative-Deepening: An Optimal Admissible Tree Search”, *Artificial Intelligence*, vol. 27, no.1, pp. 97-109, 1985.

[2] P. Hart, N. Nilsson and B. Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”, *IEEE Trans. Syst. Sci. Cyber.*, vol. 4, no. 2, pp. 100-107, 1968.

9/9/02 44