

COMP3211 Lecture 3: Agents that Search

Fangzhen Lin

Department of Computer Science and Engineering
Hong Kong University of Science and Technology

Overview

- Search problems.
- Uninformed search.
- Hueristic search.
- Game tree search.
- Local search and constraint satisfaction problems.

Search in State Spaces - Motivating Example

8-puzzle:

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

8-puzzle solving agent:

- Accessible environment: the agent knows exactly which state she is in.
- Four possible actions: move blank left, right, up, or down.
- Goal: find a sequence of actions that change the environment from the initial to the goal state.

Problem-Solving Agents

- Problem-solving agents are often goal-directed.
- Key ideas:
 - ▶ To determine what the best action is, a problem-solving agent systematically considers the expected outcomes of different possible sequences of actions that lead to some goal state.
 - ▶ A systematic search process is conducted in some representation space.
- Steps:
 - ① Goal and problem formulation
 - ② Search process
 - ③ Action execution

Problem Definition

In general, a problem consists of

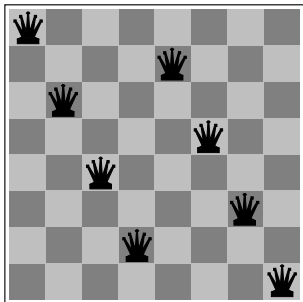
- Initial state
- Set of possible actions (or operators) and the corresponding next states
- Goal test
- Path cost function

A solution is a sequence of actions that leads to a *goal state*, i.e. a state in which the goal is satisfied.

8-Puzzle - a formulation

- States: any arrangements of the blank and the numbers 1 to 8 on the board.
- Initial state: any given one.
- Goal test: the blank in the middle, and the numbers are in order clockwise starting from the left top corner.
- Actions: move the blank left, right, up, or down.
- Path cost: the length of the path.

The 8-Queens Problem



- Goal test: 8 queens on board, none attacked
- Path cost: zero.

One possible problem formulation:

- States: any arrangement of 0 to 8 queens on board.
- Operators: add a queen to any square.

In this formulation we have 64^8 possible sequences to investigate!

The 8-Queens Problem (Cont'd)

A better one (with smaller number of states):

- States: any arrangement of 0 to 8 queens with none attacked.
- Operators: place a queen in the left-most empty column such that it is not attacked by any other queen.

Here we only have 2057 possible sequences to investigate!

This example shows that there is no unique formulation for a given problem. The right formulation makes a big difference to the size of the search space.

Searching For Solutions

Searching for a solution to a problem can be thought of as a process of building up a *search tree*:

- The root of the search tree is the node corresponding to the initial state.
- At each step, the search algorithm chooses one leaf node to expand by applying all possible actions to the state corresponding to the node.

```
function GENERAL-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

An Example Search Strategy:

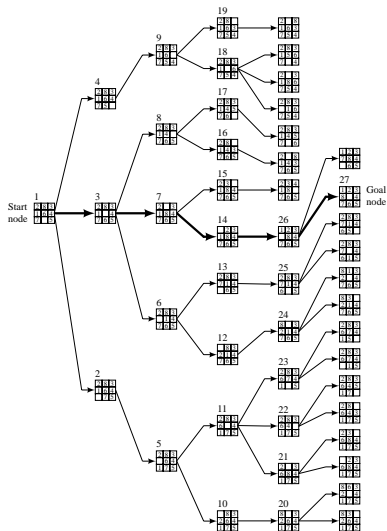
Breadth-First

- Expand first the root node.
- At each step, expand all leaf nodes.

Some properties of the breadth-first search:

- If there is a solution, then the algorithm will find it eventually.
- If there are more than one solutions, then the first one returned will be a one with the shortest length.
- The complexity (both time and space complexity) of breadth-first search is very high, making it impractical for most real world problems.

Breadth-First Search of the Eight-Puzzle



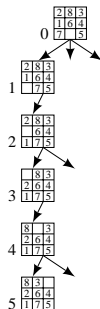
© 1998 Morgan Kaufman Publishers

Search Strategies

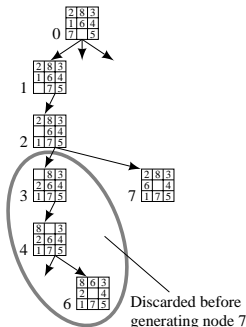
- The search strategy determines the order in which nodes in the search tree are expanded.
- Different search strategies lead to different search trees.
- Four different criteria that determine what the right search strategy is for a problem:
 - ▶ Completeness:
Is it guaranteed to find a solution if one exists?
 - ▶ Time complexity:
How long does it take to find a solution?
 - ▶ Space complexity:
How much memory is needed?
 - ▶ Optimality:
Does it find the “best” solution if several different solutions exist?
- Types:
 - ▶ Uninformed (or blind) search strategies
 - ▶ Informed (or heuristic) search strategies

Depth-First (Backtracking) Search

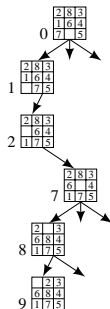
- Depth-first search generates the successors of a node one at a time;
- as soon as a successor is generated, one of its successors is generated;
- Normally, a *depth bound* is used: no successor is generated whose depth is greater than the depth bound;
- the following is a depth-first search illustration using 8 puzzle with a depth bound of 5.



(a)



(b)



(c)

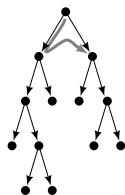
Comparing Breadth-First and Depth-First Search

In the following, b is the branching factor; d is the depth of solution; m is the depth bound of the depth first search:

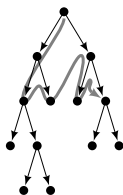
	Time	Space	Optimal?	Complete?
Breadth First	b^d	b^d	Yes	Yes
Depth First	b^m	bm	No	Yes, if $m \geq d$

Iterative Deepening Search

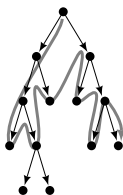
- A technique called *iterative deepening* enjoys the linear memory requirements of depth-first search while guaranteeing optimality if a goal can be found at all;
- it conducts successive depth-first search, by increasing the depth bound by 1 each time, until a goal is found:



Depth bound = 1



Depth bound = 2



Depth bound = 3



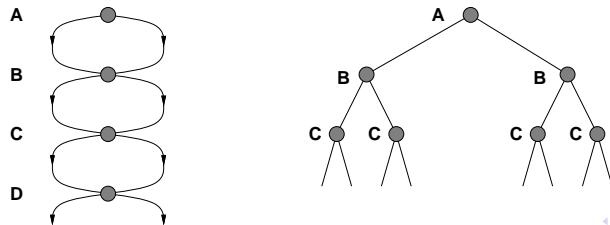
Depth bound = 4

© 1998 Morgan Kaufman Publishers

Avoiding Repeated States

- Newly expanded nodes may contain states that have already encountered before.
- There are three ways to deal with repeated states, in increasing order of effectiveness and computational overhead:
 - ▶ Do not return to the state just came from.
 - ▶ Do not create a path with cycles, that is, do not expand a node whose state has appeared in the path already.
 - ▶ Do not generate any state that was generated before.

A state space that generates an exponentially large search tree:



Heuristic Search

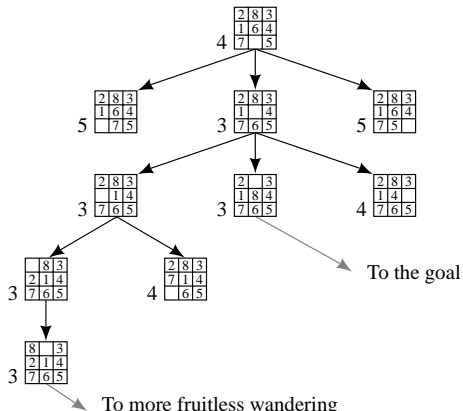
- A heuristic function is a mapping from states to numbers.
- It normally measures how far the current state is from a goal state, so the smaller the value is the closer it is from the goal.
- Heuristic or *best-first* search starts with a heuristic function and chooses nodes that have smallest value to expand.

Eight Puzzle

Consider the following heuristic function for the eight-puzzle:

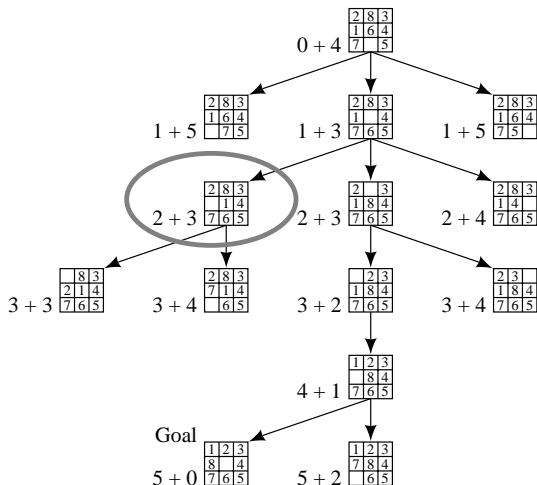
$$f(n) = \text{number of tiles out of place compared with goal}$$

Here is a search using this function (the number next to the node is its value of this function):



Eight Puzzle

The following is the search using the same heuristic function but with path cost added:



A* Search

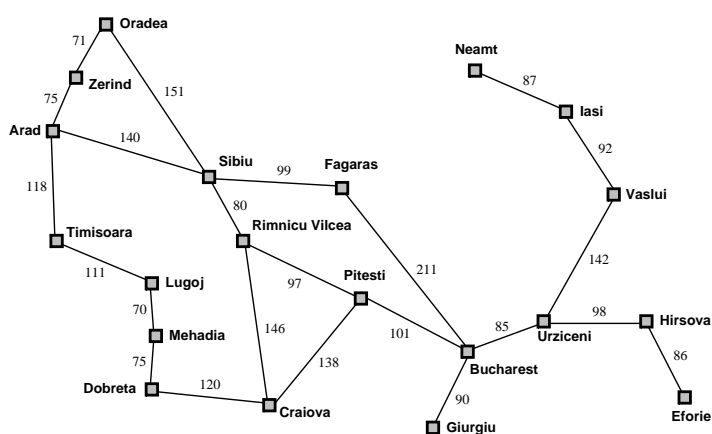
- Evaluation function:
 - ▶ Sum of:
 - ★ actual path cost $g(n)$ from the start node to node n
 - ★ estimated cost $h(n)$
 - ▶ Estimated cost of the cheapest path through node n
- Idea: Try to find the cheapest solution.

A* Search by Tree

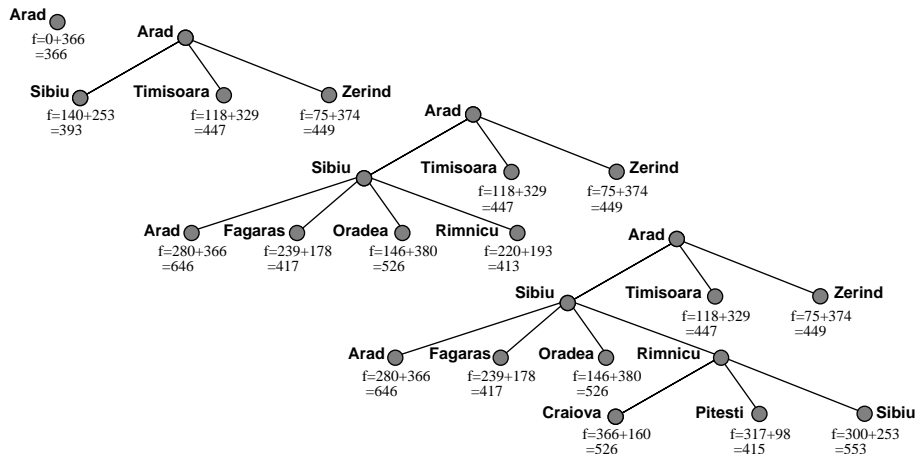
A* search by trees - check only ancestors for repeated states

- 1 create a search tree T , consisting solely of the start node, n_0 . Put n_0 on a list called *OPEN*.
- 2 If *OPEN* is empty, then exit with failure.
- 3 Select the first node on *OPEN*, and remove it from *OPEN*. Call this node n .
- 4 If n is a goal node, exit successfully with the solution corresponding to the path from root n_0 to this node.
- 5 Expand node n , generating the set M of its successors that are not already ancestors of n in G . Install these members of M as children of n in G , and add them to *OPEN*.
- 6 reorder the list *OPEN* in order of increasing $g(n) + h(n)$ values. (Ties are resolved in favor of the deepest node in the search tree.)
- 7 go to step 2.

Route Finding



A* Search for Route Finding



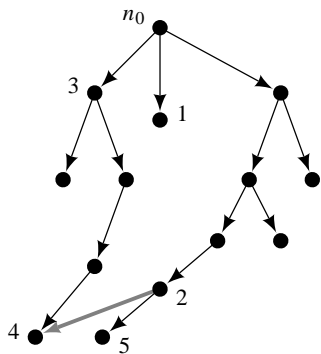
A* Search by Graph

A* search by graph - never generate any repeated states.

- 1 Create a search graph G , consisting solely of the start node, n_0 . Put n_0 on a list called *OPEN*.
- 2 create a list called *CLOSED* that is initially empty.
- 3 If *OPEN* is empty, then exit with failure.
- 4 select the first node on *OPEN*, remove it from *OPEN*, and put it on *CLOSED*. Call this node n .
- 5 If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to n_0 in G . (The pointers define a search tree and are established in step 7.)
- 6 Expand node n , generating the set M of its successors that are not already ancestors of n in G . Install these members of M as successors of n in G .

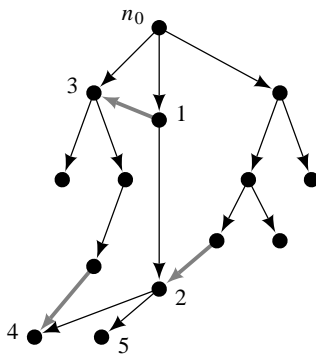
7. Establish a pointer to n from each of those members of M that were not already in G (i.e., not already in $OPEN$ or $CLOSED$). Add these members of M to $OPEN$. For each member, m , of M that was already on $OPEN$ or $CLOSED$, redirect its pointer to n if the best path to m found so far is through n . For each member of M already on $CLOSED$, redirect the pointers of each of its descendants in G so that they point backward along the best paths found so far to these descendants.
8. reorder the list $OPEN$ in order of increasing $g(n) + h(n)$ values. (Ties are resolved in favor of the deepest node in the search tree.)
9. go to step 3.

A* by Graph Link Updating



(a)

© 1998 Morgan Kaufman Publishers



(b)

Step 7

Nilsson (1998):

In step 7, we redirect pointers from a node if the search process discovers a path to that node having lower cost than the one indicated by the existing pointers. Redirecting pointers of descendants of nodes already on CLOSED saves subsequent search effort but at the possible expense of an exponential amount of computation. Hence this part of step 7 is often not implemented. Some of these pointers will ultimately be redirected in any case as the search progresses.

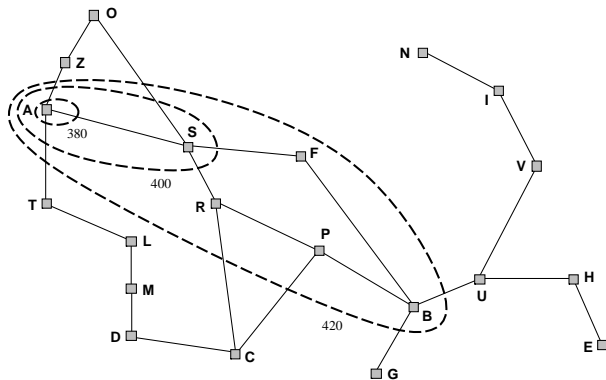
Behavior of A^*

We shall assume:

- h is *admissible*, that is, it is never larger than the actual cost.
- g is the sum of the cost of the operators along the path.
- The cost of each operator is greater than some positive amount, ϵ .
- The number of operators is finite (thus finite branching factor of search tree).

Under these conditions, we can always revise h into another admissible heuristic function so that the $f = h + g$ values along any path in the search tree is never decreasing. (*Monotonicity*.) If f^* is the cost of an optimal solution, then

- A^* expands all nodes with $f(n) < f^*$.
- A^* may expand some nodes with $f(n) = f^*$ before finding goal.

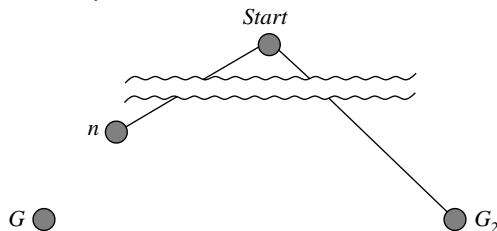


Completeness and Optimality of A^*

Under these assumptions,

- A^* is complete: A^* expands nodes in order of increasing f , there are only finite number of nodes with $f(n) \leq f^*$ is finite, so it must eventually expanded to reach a goal state.
- A^* is optimal: it will always return an optimal goal. **Proof:**
 G -optimal, G_2 -suboptimal.

It is impossible for A^* to find G_2 before G :



- It is *optimally efficient*: no other optimal algorithm is guaranteed to expand fewer nodes than A^* (Dechter and Pearl 1985).

Complexity of A^*

- Complexity:

- ▶ Number of nodes expanded is exponential in the length of the solution.
- ▶ All generated nodes are kept in memory. (A^* usually runs out of space long before it runs out of time.)
- ▶ With a good heuristic, significant savings are still possible compared to uninformed search methods.
- ▶ Admissible heuristic functions that give higher values tend to make the search more efficient.

- Memory-bounded extensions to A^* :

- ▶ Iterative deepening A^* (IDA^*)
- ▶ Simplified memory-bounded A^* (SMA^*)

Heuristic Functions for 8-Puzzle

- Some possible candidates (admissible heuristic functions):
 - Number of tiles that are in the wrong position (h_1)
 - Sum of the city block distances of the tiles from their goal positions (h_2)
- Comparison of iterative deepening search, A^* with h_1 , and A^* with h_2 (averaged over 100 instances of the 8-puzzle, for various solution lengths d):

d	Search Cost			Effective Branching Factor		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	—	1301	211	—	1.45	1.25
18	—	3056	363	—	1.46	1.26
20	—	7276	676	—	1.47	1.27
22	—	18094	1219	—	1.48	1.28
24	—	39135	1641	—	1.48	1.26

- We see that h_2 is better than h_1 .
- In general, it is always better to use a heuristic function with higher values, as long as it is admissible, i.e. does not overestimate: h_2 *dominates* (is better than) h_1 because for all node n , $h_2(n) \geq h_1(n)$.

Inventing Heuristic Functions

Questions:

- How might one have come up with good heuristics such as h_2 for the 8-puzzle?
- Is it possible for a computer to mechanically invent such heuristics?

Relaxed Problem. Given a problem, a relaxed problem is one with less restrictions on the operators.

Strategy. use the path cost of a relaxed problem as the heuristic function for the original problem.

Inventing Heuristic Function - 8-Puzzle

Example. Suppose the 8-puzzle operators are described as:

A tile can move from square A to B if A is adjacent to B and B is blank.

We can then generate three relaxed problems by removing some of the conditions:

- (a) A tile can move from square A to B if A is adjacent to B.
- (b) A tile can move from square A to B if B is blank.
- (c) A tile can move from square A to B.

Using (a), we get h_2 . Using (c), we get h_1 .

ABSOLVER (Prieditis 1993) is a computer program that automatically generates heuristics based on this and other techniques.

Games - An Example

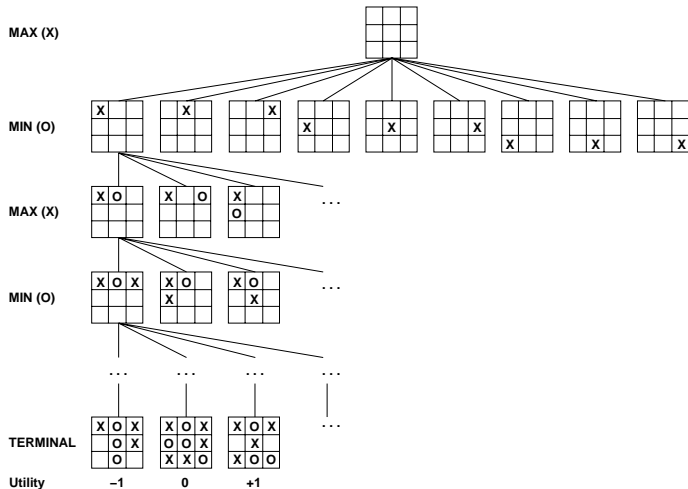
Tic-Tac-Toe:

- A board with nine squares.
- Two players: “X” and “O”; “X” moves first, and then alternate.
- At each step, the player choose an unoccupied square, and mark it with his/her name. Whoever gets three in a line wins.

Games as Search Problems

- Mainly two-player games are considered.
- Two-player game-playing problems require an agent to plan ahead in an environment in which another agent acts as its opponent.
- Two sources of uncertainty:
 - ▶ Opponent: The agent does not know in advance what action its opponent will take.
 - ▶ Time-bounded search: Time limit has to be imposed because searching for optimal decisions in large state spaces may not be practical. The “best” decisions found may be suboptimal.
- Search problem:
 - ▶ Initial state: initial board configuration and indication of who makes the first move.
 - ▶ Operators: legal moves.
 - ▶ Terminal test: determines when a terminal state is reached.
 - ▶ Utility function (payoff function): returns a numeric score to quantify the outcome of a game.

Game Tree for Tic-Tac-Toe

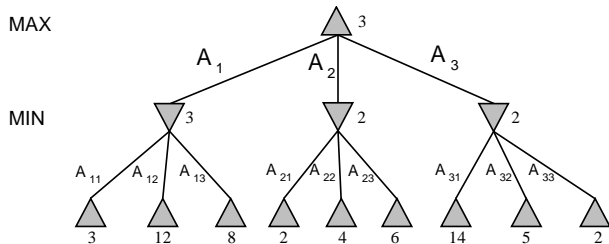


Minimax Algorithm With Perfect Decisions

Minimax Algorithm (With Perfect Decisions) Assume the two players are: MAX (self) and MIN (opponent). To evaluate a node n in a game tree:

- 1 Expand the entire tree below n .
- 2 Evaluate the terminal nodes using the given utility function.
- 3 Select a node that has not been evaluated yet, and all of its children have been evaluated. If there are no such node, then return.
- 4 If the selected node is on at which the MIN moves, assign it the minimum of the values of its children. If the selected node is on at which the MAX moves, assign it the maximum of the values of its children. Return to step 3.

MAX tries to *maximize* the utility, assuming that MIN will act to *minimize* it.



Imperfect Decisions

- Minimax algorithm with perfect decisions:
 - ▶ No time limit is imposed.
 - ▶ The complete search tree is generated.
- It is often impractical to make perfect decisions, as the time and/or space requirements for complete game tree search (to terminal states) are intractable.
- Two modifications to minimax algorithm with perfect decisions:
 - ▶ Partial tree search:
Terminal test is replaced by a *cutoff test*.
 - ▶ Evaluation function:
Utility function is replaced by a heuristic *evaluation function*.

Minimax Algorithm (With Imperfect Decisions) Assume the two players are: MAX (self) and MIN (opponent). To evaluate a node n in a game tree:

- 1 Expand the tree below n according to the partial tree search.
- 2 Evaluate the leaf nodes using the given evaluation function.
- 3 Select a node that has not been evaluated yet, and all of its children have been evaluated. If there are no such node, then return.
- 4 If the selected node is on at which the MIN moves, assign it the minimum of the values of its children. If the selected node is on at which the MAX moves, assign it the maximum of the values of its children. Return to step 3.

Evaluation Functions

- An evaluation function returns an *estimate* of the *expected utility* of the game from a given position.
- Requirements:
 - ▶ Computation of evaluation function values is efficient.
 - ▶ Evaluation function agrees with utility function on terminal states.
 - ▶ Evaluation function accurately reflects the chances of winning.
- Most game-playing programs use a *weighted linear function*:

$$w_1 f_1 + w_2 f_2 + \cdots w_n f_n$$

where the f 's are the features (e.g. number of queens in chess) of the game position, and w 's are the weights that measure the importance of the corresponding features.

- *Learning* good evaluation functions *automatically* from past experience is a promising new direction.

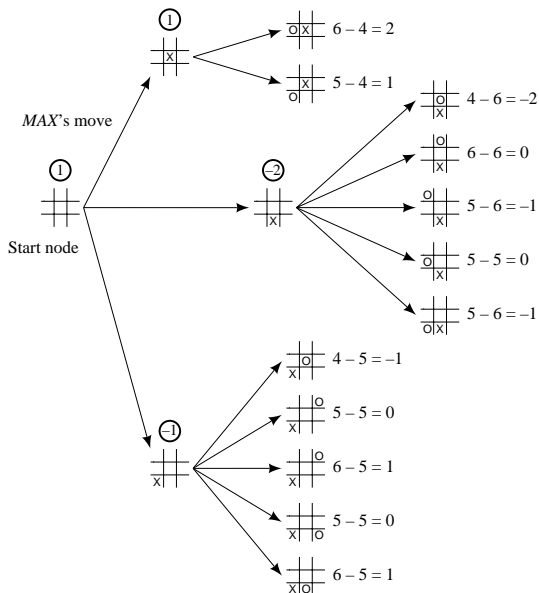
Tic-Tac-Toe

Evaluation function, $e(p)$, for Tic-Tac-Toe

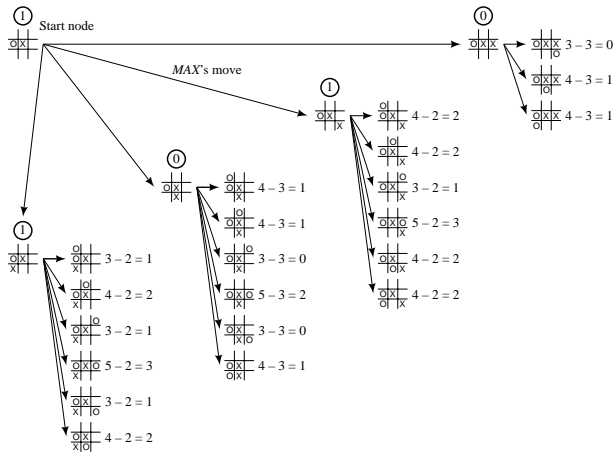
- If p is not a winning position for either player, then $e(p) = (\text{the number of complete rows, columns, or diagonals that are still open for MAX}) - (\text{the number of complete rows, columns, or diagonals that are still open for MIN})$.
- If p is a win for MAX, then $e(p) = \infty$; if p is a win for MIN, then $e(p) = -\infty$.

The next three slides illustrate Minimax search with imperfect decisions. Notice that we have eliminated symmetric positions.

The First Stage of Search

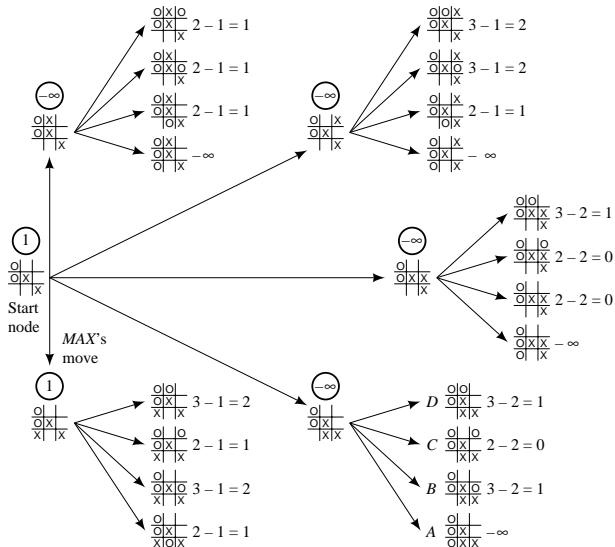


The Second Stage of Search



© 1998 Morgan Kaufman Publishers

The Last Stage of Search

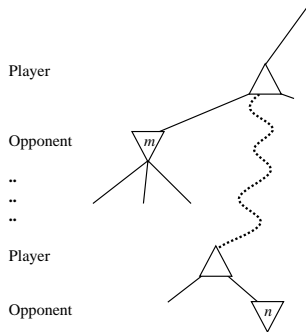


Partial Tree Search

- Different approaches:
 - ▶ Depth-limited search
 - ▶ Iterative deepening search
 - ▶ Quiescent search
- Quiescent positions are positions that are not likely to have large variations in evaluation in the near future.
- Quiescent search:
 - ▶ Expansion of nonquiescent positions until quiescent positions are reached.

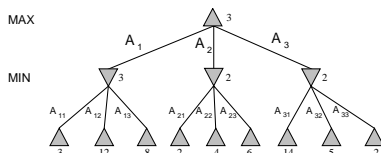
Pruning

- Pruning: The process of eliminating a branch of the search tree from consideration without actually examining it.
- General idea: If m is better than n for Player, then n will never be reached in actual play and hence can be pruned away.

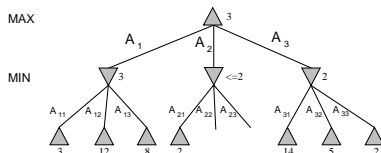


Alpha-Beta Pruning

- Minimax search without pruning:



- Minimax search with alpha-beta pruning:



- Effectiveness of alpha-beta pruning depends on the order in which successor nodes are examined.

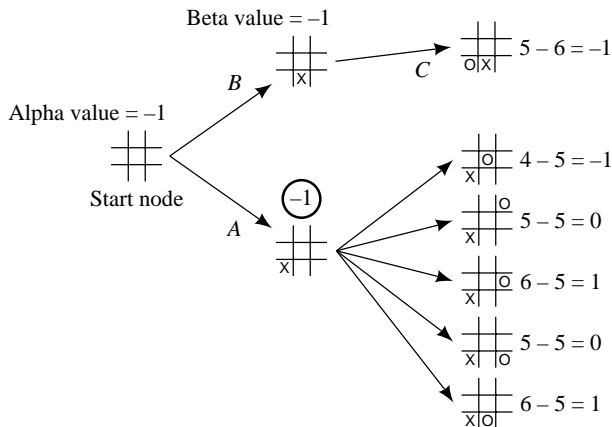
Alpha-Beta Search Algorithm - Informal Description

To evaluate a MAX node in a game tree (we shall call the value assigned to a MAX node its alpha value, and that to a MIN node its beta value):

- ❶ Expand the node depth-first until a node that satisfies the cutoff test is reached.
- ❷ Evaluate the cutoff node.
- ❸ Update the values of all the nodes that have so far been expanded according to Minimax algorithm, and using the following pruning strategy:
 - ▶ prune all children of any MIN node whose beta value is \leq the alpha value of any of its MAX ancestors.
 - ▶ prune all children of any MAX node whose alpha value is \geq the beta value of any of its MIN ancestors.
- ❹ Backtrack to a node that has not been pruned, and go back to step 1. If there are no such node to backtrack to, then return with the value assigned to the original node.

Example

Part of the first state of search in Tic-Tac-Toe using Alpha-Beta search.



© 1998 Morgan Kaufman Publishers

Alpha-Beta Search Algorithm

The above informal description corresponds to calling the following function

MAX-VALUE(*node*, *game*, $-\infty$, ∞):

function MAX-VALUE(*state*, *game*, α , β) **returns** the minimax value of *state*

inputs: *state*, current state in game

game, game description

α , the best score for MAX along the path to *state*

β , the best score for MIN along the path to *state*

if CUTOFF-TEST(*state*) **then return** EVAL(*state*)

for each *s* **in** SUCCESSORS(*state*) **do**

$\alpha \leftarrow \text{MAX}(\alpha, \text{MIN-VALUE}(s, \text{game}, \alpha, \beta))$

if $\alpha \geq \beta$ **then return** β

end

return α

function MIN-VALUE(*state*, *game*, α , β) **returns** the minimax value of *state*

if CUTOFF-TEST(*state*) **then return** EVAL(*state*)

for each *s* **in** SUCCESSORS(*state*) **do**

$\beta \leftarrow \text{MIN}(\beta, \text{MAX-VALUE}(s, \text{game}, \alpha, \beta))$

if $\beta \leq \alpha$ **then return** α

end

return β

Analyses of Game-Playing Algorithm

- The effectiveness of alpha-beta pruning depends on the ordering in which successors are generated.
- Assuming optimal ordering, alpha-beta pruning can reduce the branching factor from b to \sqrt{b} (Knuth and Moore 1975).
- The result also assumes that all nodes have the same branching factor, that all paths reach the same fixed depth limit, and that the leaf (cutoff node) evaluations are randomly distributed.
- All the game-playing algorithms assume that the opponent plays optimally.

State of Art

- Chess: Deep Blue (Benjamin, Tan, *et al.*) defeated world chess champion Gary Kasparov on May 11, 1997.
- Checker: Chinook (Schaeffer *et al.*) became the world champion in 1994.
- Go: US\$2,000,000 prize for the first program to defeat a top-level player.

Alternative Search Formulations

- Assignment problem or constraint satisfaction.
- Function optimization.

Assignment Problems

Problem definition:

- A finite set of variables and their domains.
- A finite set of conditions on these variables.
- A solution is an assignment to these variables that satisfies all the conditions.

Example: 8-queens problem:

- Variables: q_1, \dots, q_8 , the position of a queen in column $1, \dots, 8$.
- Domain: the same for all variables, $\{1, \dots, 8\}$.
- Constraints: $q_1 - q_2 \neq 0, |q_1 - q_2| \neq 1, \dots$

Constructive Methods

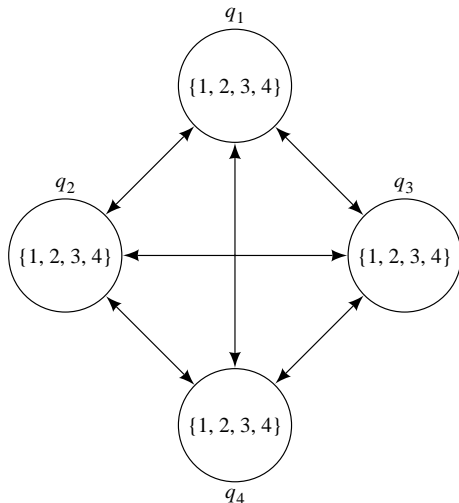
Constructive methods try to find a solution using search strategies, especially depth-first search, to find a solution. A technique called *constraint propagation* can be used to prune the search space:

If there is a constraint involving both variables i and j , then knowing the values of i may eliminate possible values for j .

A *constraint graph* is often used to do constraint propagation: a node is labelled by a variable and the domain that the variable can take, and there is an edge from node i to j if the variable i and j are constrained.

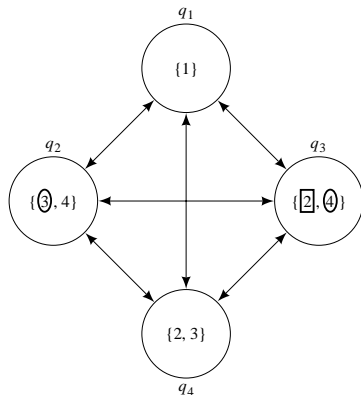
Running Example: 4-Queens Problem

Constraint graph:



Running Example: 4-Queens Problem

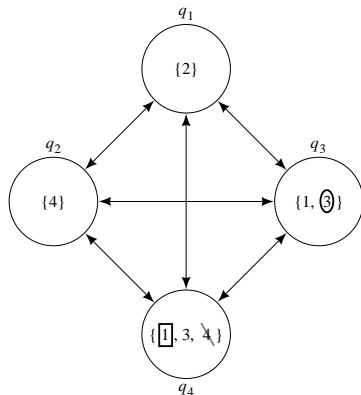
Constraint graph with $q_1 = 1$:






- ☐ Values eliminated by first making arc (q_2, q_3) and then arc (q_3, q_2) consistent
- ☐ Value eliminated by next making arc (q_3, q_4) consistent

Running Example: 4-Queens Problem

Constraint graph with $q = 2$:



-  Value eliminated by first making arc (q_3, q_2) consistent
-  Value eliminated by next making arc (q_4, q_3) consistent
-  Value eliminated by next making arc (q_4, q_2) consistent

Heuristic Repair

- Constructive methods starts with empty assignment, and build up a solution gradually by assigning values to variables one by one.
- *Heuristic repair* starts with a proposed solution, which most probably does not satisfy all constraints, and repair it until it does.
- An often used, so-called *min-conflicts* repair method by Gu will select a variable for adjust, and find a value that minimizes conflicts.

8-Queens By Min-Conflicts

2					x		
0							
x2				x			
1							x
3		x					
2	x						
4						x	
1			x				

Move
(1,3) to (1,2)

	2				x		
x	2						
	2			x			
	2						x
	3	x					
	x3						
	1					x	
	1	x					

Move (2,6)
to (2,7)

	2				x		
x	1						
	2			x			
	2						x
	x1						
	1						
x	3					x	
	2	x					

No change

		2		x			
x		2					
		2	x				
		4				x	
	x	3					
		1					
x		2				x	
		x1					

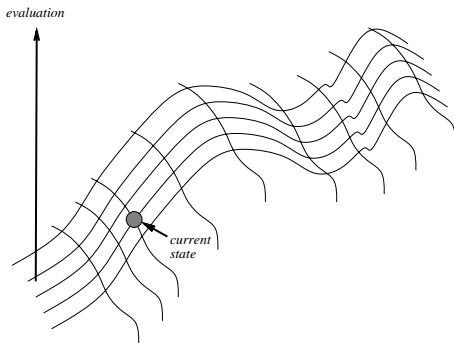
Search as Function Maximization

- Function maximization problems: find a x such that $Value(x)$ is maximal.
- Examples: 8-queens problem
 - ▶ find a state in which none of the 8 queens are attacked.
 - ▶ design a function on the states such that its value is maximal when none of the queens are attacked.
 - ▶ one such function can be: $Value(n) = 1/e^{k(n)}$, where $k(n)$ is the number of queens that are attacked.
- Another example: VLSI design - find a layout that satisfies the constraints.

Hill-Climbing

Basic ideas:

- Start at the initial state.
- At each step, move to a next state with highest value.
- It does not maintain the search tree and does not backtrack - it keeps only the current node and its evaluation.



Hill-Climbing Search Algorithm

```
function HILL-CLIMBING(problem) returns a solution state
  inputs: problem, a problem
  static: current, a node
           next, a node

  current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  loop do
    next  $\leftarrow$  a highest-valued successor of current
    if VALUE[next] < VALUE[current] then return current
    current  $\leftarrow$  next
  end
```

Simulated Annealing

- Hill-climbing can get easily stuck in a local maximum.
- One way to avoid getting stuck in a local maximum is by using *simulated annealing*.
- The main idea:
 - ▶ at each step, instead of picking the best move, it picks a random move.
 - ▶ if the move leads to a state with higher value, then execute the move.
 - ▶ otherwise, execute the move with certain probability that becomes smaller as the algorithm progresses.
 - ▶ the probability is computed according to a function (*schedule*) that maps time to probabilities.
- The idea comes from the process of annealing - cooling metal liquid gradually until it freezes.
- It has been found to be extremely effective in real applications such as factory scheduling.