UNIVERSITY OF LOUISVILLE
J.B. SPEED SCHOOL OF ENGINEERING

Computer Engineering and Computer Science Department

CECS545-Artificial Intelligence

Search 2

Dr. Roman Yampolskiy

Practice Exercises from Chapter 3

- Solve the following problems from the book:
- 3.10, 3.14, 3.15 (a, b, c)

Exercise 3.10

- Define in your own words the following terms:
  - State, state space, search tree, search node, goal, action, successor function (transition model), and branching factor.

Exercise 3.14

3.14 Which of the following are true and which are false? Explain your answers.

a. Depth-first search always expands at least as many nodes as A* search with an admissible heuristic.

b. $h(n) = 0$ is an admissible heuristic for the 8-puzzle.

c. A* is of no use in robotics because percepts, states, and actions are continuous.

d. Breadth-first search is complete even if zero step costs are allowed.

e. Assume that a rook can move on a chessboard any number of squares in a straight line, vertically or horizontally, but cannot jump over other pieces. Manhattan distance is an admissible heuristic for the problem of moving the rook from square A to square B in the smallest number of moves.

Exercise 3.15 (A, B, C)

3.15 Consider a state space where the start state is number 1 and each state k has two successors: numbers $2k$ and $2k + 1$.

a. Draw the portion of the state space for states 1 to 15.

b. Suppose the goal state is 11. List the order in which nodes will be visited for breadth-first search, depth-limited search with limit 3, and iterative deepening search.

c. How well would bidirectional search work on this problem? What is the branching factor in each direction of the bidirectional search?

CECS545-Artificial Intelligence

Search Cont.

Dr. Roman Yampolskiy

Problem Solving using State - Space Search

- The trial -and -error approach to problem solving

Problems are solved by searching among alternative choices

Try alternatives until you find a solution

Many problems can be represented as a set of states and a set of rules of how one state is transformed to another.

- The problem is how to reach a particular goal state, starting from an initial state and using the state traversing rules.

Game-Playing, Theorem Proving, Diagnosis, Query Optimization, …

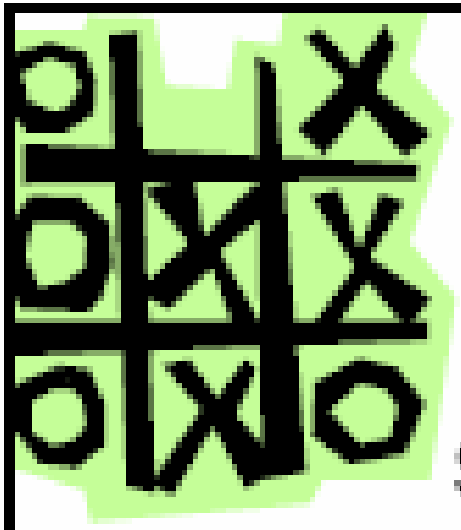
Search Classification

- • Strategies for finding a goal. No concern of the cost associated with it.

- No information of the search space (e.g. depth first)

- Some information on the structure of the search space such as estimates to the distance to the goal (e.g. best first)

- • Strategies for finding a minimum cost path to the goal (e.g. branch and bound)
- • Strategies for finding a goal in presence of adversaries such as game playing (e.g. A-B pruning)

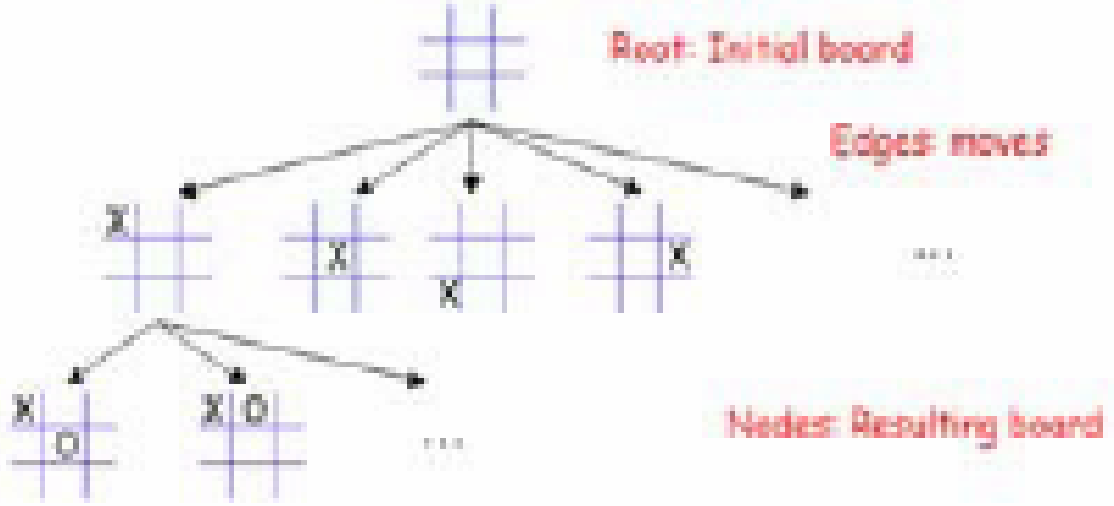


Example: Tic -Tac-Toe

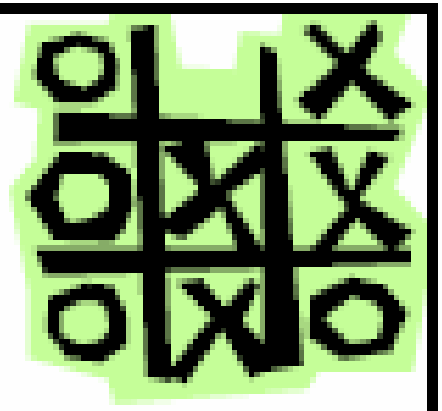
State Space Graph

Nodes: solution states of the problem

Edges: steps in the problem-solving process



Tic -tac- toe (cont.)



- Why define a state space for tic -tac-toe?
  - Each path gives every possible game

  - Given a board configuration, we can find all of our possible next moves, and from these, all of our opponents, …, until we end in a final state

  - We choose the move that maximizes our wins and minimizes our losses

State Space Search Graphs

- Typically have a set of start states (e.g., the empty board)

And, a set of goal states:

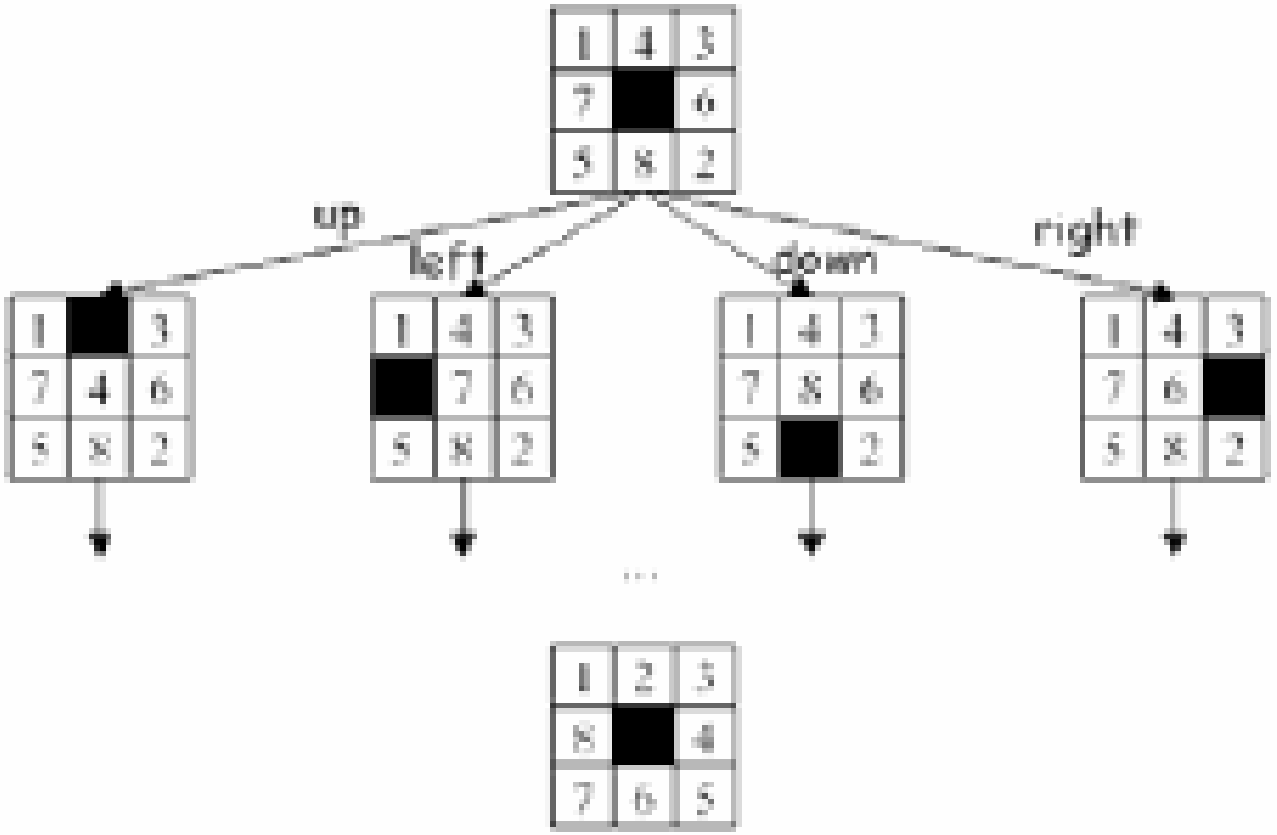
  – Either some property of states (like a winning tic-tac-toe board)

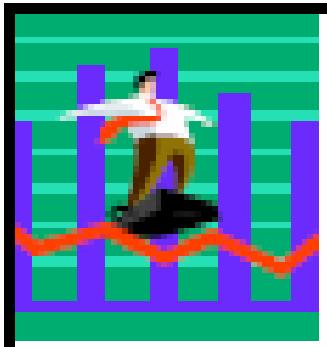
  – Or, a property of the paths

A solution path begins at a start node and ends at a goal node

  – There may be zero or more solution paths

The 8 -Puzzle

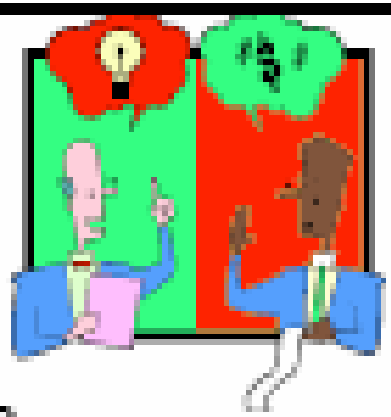

1 4 3
7 6
5 8 2
up
left
down
right
1 3
7 4 6
5 8 2
1 4 3
7 6
5 8 2
1 4 3
7 8 6
5 2
1 4 3
7 6
5 8 2
...
1 2 3
8 4
7 6 5



Generating States and Search

- States are generated by applying operators to the existing state
  - The "next move" operator for tic-tac-toe
  - The up, down, left, right operators
- We search states to find problem solutions
  - Thus, the search procedure applies the state generation operators

Search Strategies



- Measuring problem-solving performance:
  - does it find a solution at all?
  - is it a good solution (one with a low path cost)?
  - what is the search cost associated with the time and memory required to find a solution?
- The total cost of the search is the sum of the path cost and the search cost.
- Search Strategy is evaluated in four criteria:
  - completeness
  - time complexity
  - space complexity
  - optimality/admissibility
  - Informed search (heuristic search) versus uninformed search (blind search)

Search Strategies

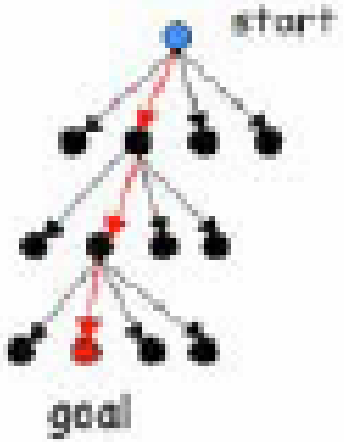
- Data -driven search
  - Also called forward –chaining
  - Begin with the known facts of the problem
  - Apply the legal moves to the facts to generate new facts
  - Repeat until we find a path to a goal state
  - This is what we did in 8 -puzzle and tic-tac-toe

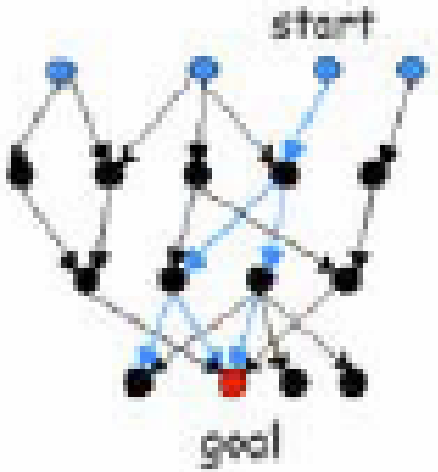
Search Strategies (cont.)

- Goal -Driven search
  - Also called backward –chaining
  - Like "trying to solve a maze by working backward from the finish to the start"
  - Start with the goal state
  - Find the states (subgoals) from which legal moves would generate the goal
  - Repeat until we get to the known facts of the problem
  - Prolog performs goal -driven search…

Is Data - or Goal-Driven Search Better?

- Depends on:
  - The structure of the search space
  - How well it can be pruned
  - Nature of the problem



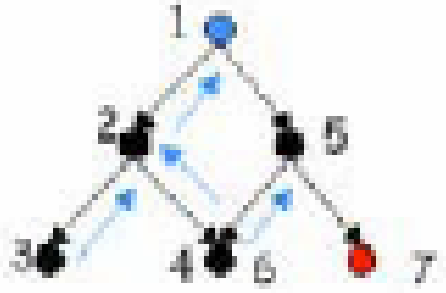


Is Data - or Goal-Driven Search Better?

- Goal-Driven search is recommended if:
  - A goal or hypothesis is given in the problem statement or can easily be formulated.
  - There is a large number of rules that match the facts of the problem and thus produce an increasing number of conclusions or goals.
  - Problem data are not given but must be acquired by the problem solver.
- Data-Driven search is recommended if:
  - All or most of the data are given in the initial problem statement.
  - There is a large number of potential goals, but there are only a few ways to use the facts and given information of the particular problem.
    - It is difficult to form a goal or hypothesis.

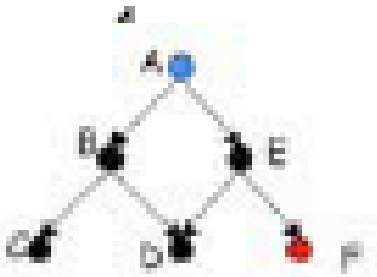
Search Algorithms: Backtracking

- Search begins at the start state
- Pursues a path until it finds a goal or a dead end
- When it finds a dead -end it "backtracks"



Backtracking Example

After Iteration	Curr	StateList	NewSL	DeadEnds
0	A	[A]	[A]	[ ]
1	B	[B, A]	[B, E, A]	[ ]
2	C	[C,B,A]	[C,D,B,E,A]	[ ]
3	D	[D,B,A]	[D,B,E,A]	[C]
	E	[E,A]	[E,A]	[C,
	F	[F,E,A]	[F,E,A]	[C,
6	F	[F,E,A]		



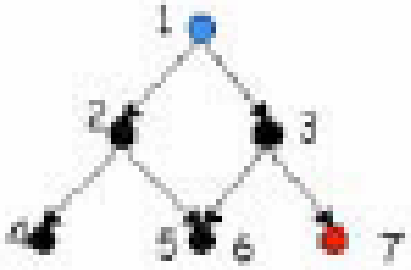
More on Backtrack

- Can be goal -driven by switching goal and start

- Backtrack keeps track of:
  - Unprocessed states (NewSL)
  - Dead End states (to prevent searching useless paths)
  - The solution path (StateList)
  - Unexamined states (to prevent cycles)

Breadth -First Search

- Consider every node at each level of the graph before going deeper into the space
- Guaranteed to find the shortest path



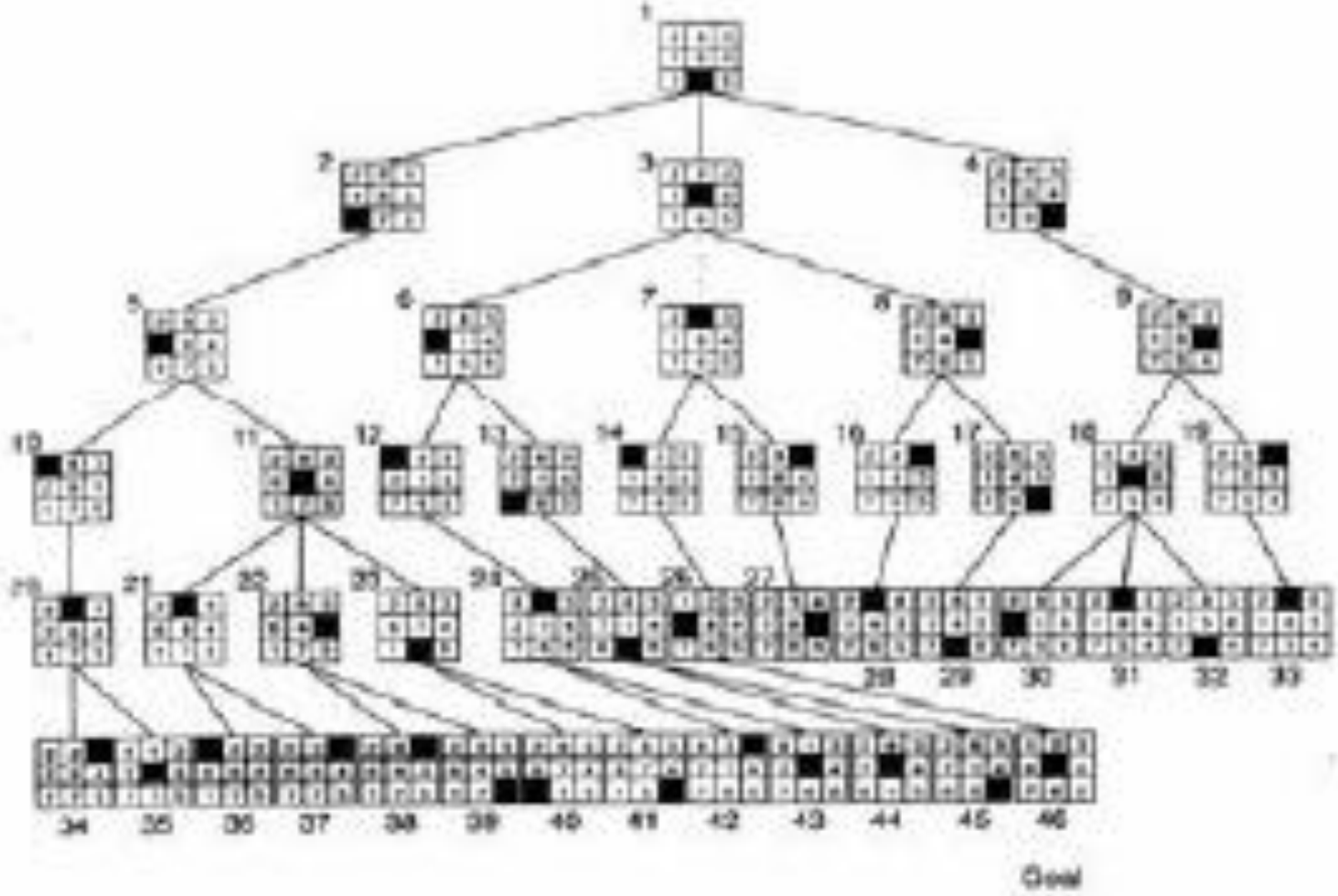
Breadth -First Search

- In breadth-first, all the siblings of a node are explored before their children are expanded.
- 1. Form a one-element queue consisting of the root node.
- 2. Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node.
- 2a. If the first element is not the goal node, remove the first element from the queue and add the first element's children, if any, to the back of the queue.

Breadth -First Search Algorithm

```
1   Open = [Start]; Closed = [ ];
2   while Open  [ ] do
3       S = head(Open)
4       remove first element from Open
5       if S = goal
6          return SUCCESS
7       else
8          generate children of S
9          prepend S to Closed
10         discard children of S if already on Open or Closed
11        append remaining children on Open
12  end while
13  return FAIL
```

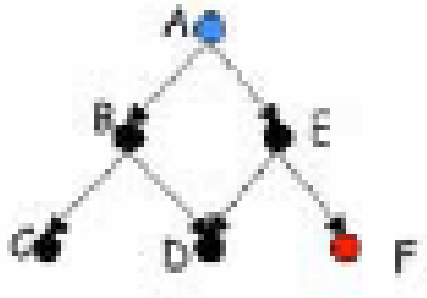
Breadth -First Search Example



Breadth -First Search Example

After Iteration	Open	Closed
0	[A]	[ ]
1	[B,E]	[A]
2	[E,C,D]	[B,A]
3	[C,D,F]	[E,B,A]
4	[D,F]	[C,E,B,A]
5	[F]	[D,E,B,A]
6	[ ]	

We are done!



Note: We don't have the path!

We can track paths by storing paths (via parents) in closed…

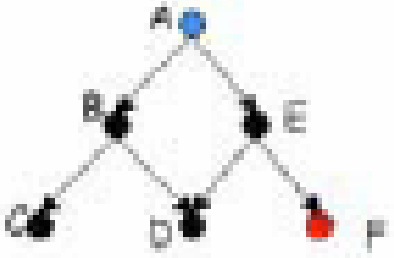
Depth -First Search: Simplified Backtrack

```
1    Open = [Start]; Closed = [ ];
2    while Open ≠ [ ] do
3        S = head(Open)
4        remove first element from Open
5        if S = goal
6          return SUCCESS
7        else
8          generate children of S
9          prepend S to Closed
10         discard children of S if already on Open or Closed
11         prepend remaining children of S on Open
12   end while
13   return FAIL
```

Depth -First Search Example

After Iteration	Open	Closed
0	[A]	[ ]
1	[B,E]	[A]
2	[C,D,E]	[B,A]
3	[D,E]	[C,B,A]
4	[E]	[D,C,B,A]
5	[F]	[E,D,C,B,A]
6	[ ]	We are done!

Note: Could store path too…



Depth -First vs. Breadth -First Search

- **They Both:**
  - Find a solution if one exists (and if the state isn't too big…)

Depth-First

- Not guaranteed to find shortest path
- Can get stuck on an infinite path

Breadth –First

- Takes up more memory…
- … Must remember the unexpanded nodes at a level, which could become very large as you go deeper into the tree/graph
- Depth -First remembers just the path

Iterative Deepening

- Place a Depth-Bound on Depth-First Search:

  – Start with a breadth -first search, but examine children using depth-first search down to a fixed level

  – Good for large spaces (like in chess) where you can't exhaustively search

Brute- force search

- A brute force method: all paths are explored, and from the ones leading to a goal, the optimum (minimal cost) is chosen
- No information about the search space is assumed
- This strategy is effective for small (narrow and shallow) search spaces
- A regular tree with a branching factor of B and depth L:

 - number of nodes to explore:

 - number of paths to explore:

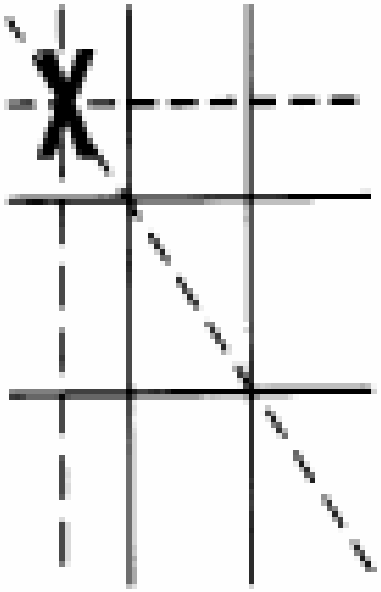
 - example: B=5 and L=10:

- Brute-force search techniques (i.e., pure depth-first & pure breadth-first) may fail to find a solution within any practical length of time

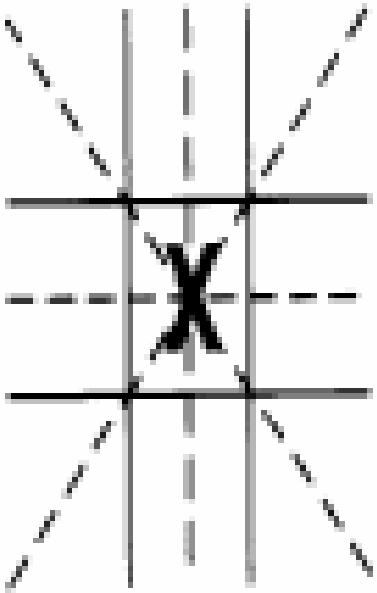
Heuristics

- An informed guess of the next step to be taken in solving a problem would prune the search space
- A heuristic is any rule or method that provides guidance in decision making
- A problem may not have an exact solution because of the inherent ambiguities in the problem statement or available data
- Heuristics are viewed as 'rules of thumb' that domain experts could use to generate good solutions without exhaustive search (rule-based expert systems).
- Heuristic is the opposite of algorithmic
- A heuristic may find a sub-optimal solution or fail to find a solution since it uses limited information
- In search algorithms, heuristic refers to a function that provides an estimate of solution cost

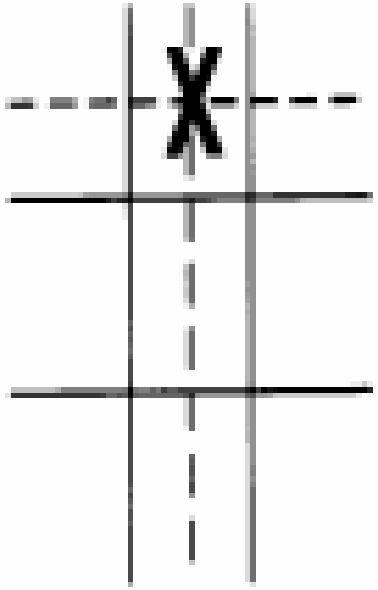
A heuristic for Tic-Tac-Toe



Three wins through a corner square

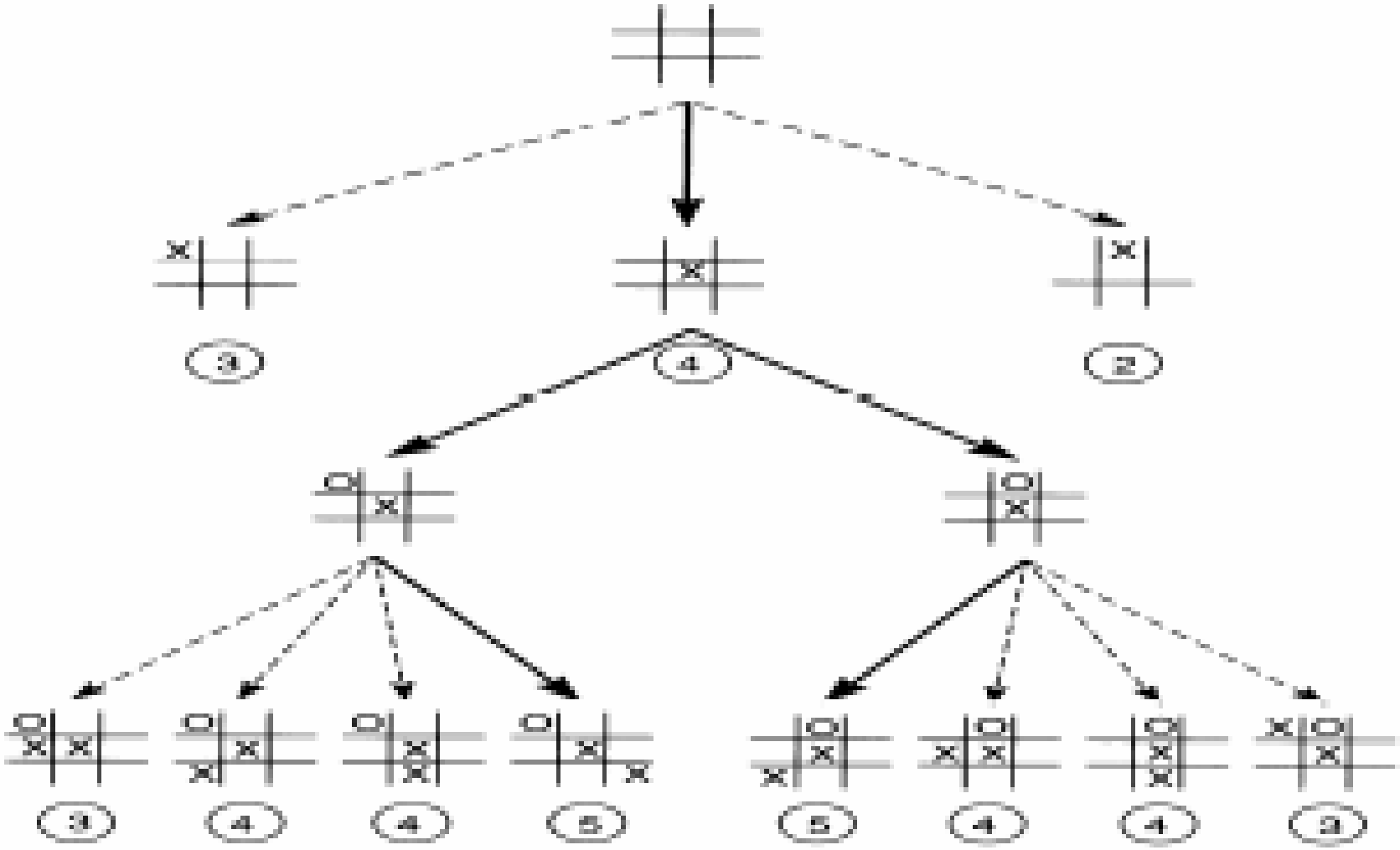


Four wins through the center square



Two wins through a side square

State space search using heuristic



Hill climbing search

- Uses local estimates of the distance to the goal and expands the node with the smallest estimate
- Retains no memory: queue length of 1
- If a node has no children, then the search terminates
- The goal is deemed to have been reached if the cost of all the children is larger than the cost of the parent
- To conduct a hill climbing search (similar to depth-search!):

1. Form a one-element queue consisting of the root node.
2. Until the queue is empty or the goal has been reached, determine if the first element of the queue is the goal node.

2a. If the first element of the queue is not the goal node, remove the first element from the queue, *sort the first element's children, if any, by estimated remaining distance (cost)*, and add the first element's lowest-cost child, if any, to the queue.

Hill climbing discussion

- Suitable for problems with adjustable parameters and a quality measurement associated with these parameters
- Instead of an explicit goal, the procedure stops when a node is reached where all the node's children have lower quality measurements
- Hill climbing performs well if the distance estimate (quality measurement) is locally consistent. That is, the node with the shortest distance estimate will eventually lead to the goal
- Hill climbing may encounter problems:

- stuck at local optimum

- ignore global optimum

Best first search

- Similar to hill climbing, it uses a cost function to estimate the distance from the goal
- But it remembers the unexplored nodes
- The children of the currently explored node and previously unexplored nodes are sorted
- To conduct a best-first search (similar to hill climbing):

1. Form a one-element queue consisting of the root node.
2. Until the queue is empty or the goal has been reached, determine if the first element of the queue is the goal node.

2a. If the first element of the queue is not the goal node, remove the first element from the queue, add the first element's children, if any, to the queue, and *sort the entire queue by estimated remaining distance (cost).*

Beam search

- Similar to breadth-first, it explores the search tree level by level
- But it keeps only the best w nodes from each level
- To conduct a beam search:

1. Form a one-element queue consisting of the root node.
2. Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node.
   2a. If the first element is not the goal node, remove the first element from the queue, find the first element's children, if any.
   2b. Sort all the children found so far, by estimated remaining distance.
   Add the best w children to the back of the queue.

- Beam search is very efficient since it explores only w nodes at each level. However, it may fail due to local minima

Branch and bound

- Branch and bound uses the costs of the already formed partial paths to guide the search
- Expand the partial path with the minimum cost
- Even if the goal is found, keep expanding until all the partial paths have costs which are greater than or equal to the minimal cost of the solution path found so far
- To conduct a branch and bound search:

  1. Form a queue of partial paths. Let the initial queue consist of the zero-length,zero-step path from the root node to nowhere.

  2. Until the queue is empty or the goal has been reached, determine if the first path in the queue is the goal node.

     2a. If the first path does not reach the goal node:

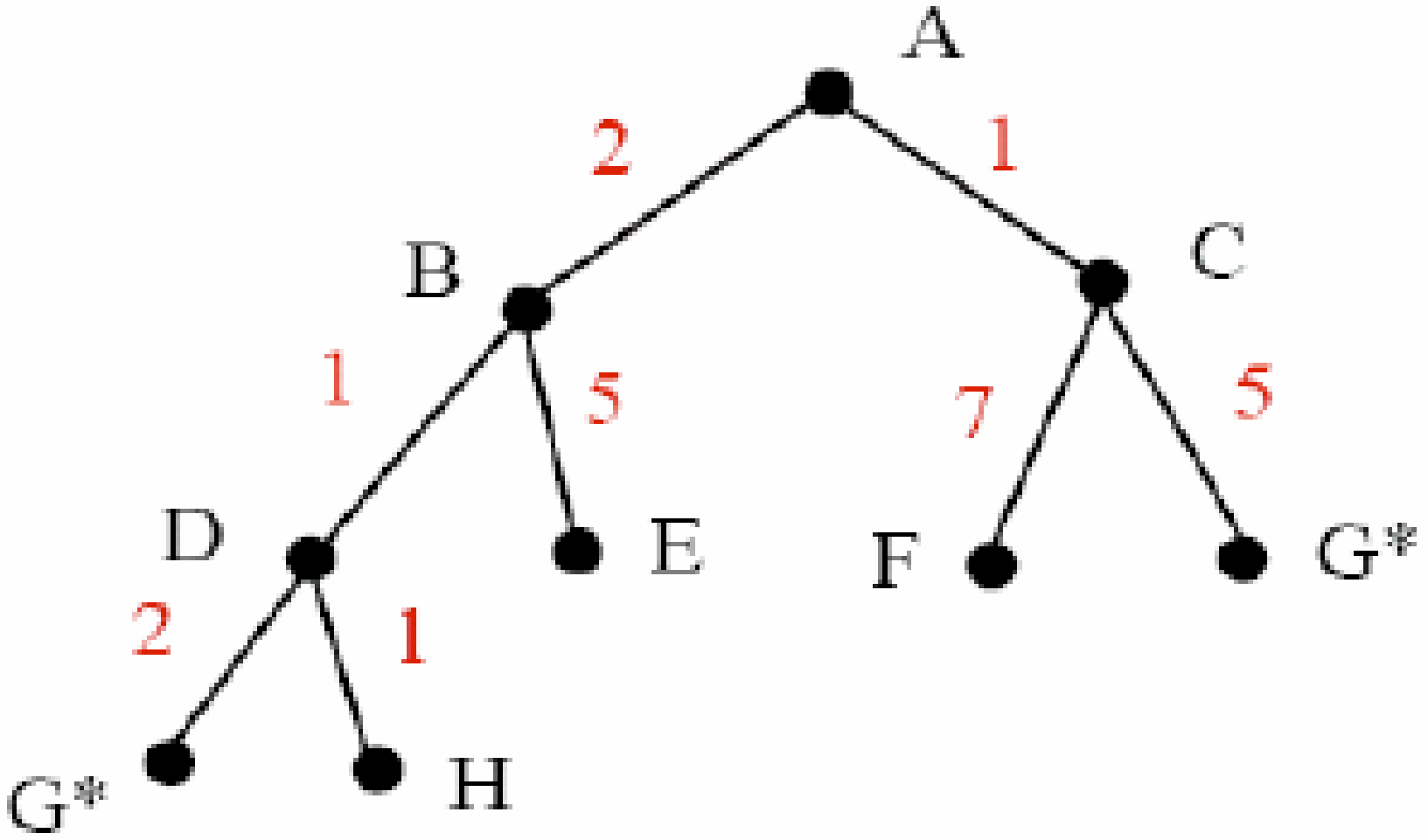
        2a1. Remove the first path from the queue.

        2a2. Form new paths from the removed path by extending one step.

        2a3. Add the new paths to the queue.

     2a4. Sort the queue by increasing cost accumulated so far.

Branch and bound: example



Branch and bound with dynamic programming

- To conduct a branch and bound search with dynamic programming:

1. Form a queue of partial paths. Let the initial queue consist of the zero-length,zero-step path from the root node to nowhere.
2. Until the queue is empty or the goal has been reached, determine if the first path in the queue reaches the goal node.

   2a. If the first path does not reach the goal node:

   2a1. Remove the first path from the queue.

   2a2. Form new paths from the removed path by extending one step.

   2a3. Add the new paths to the queue.

   2a4. Sort the queue by increasing cost accumulated so far.

   2a5. *If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.*

Branch and bound with underestimates

- Pure branch and bound uses information of the search that has been obtained so far
- One may also use information about the cost of the remaining steps to the goal
- If at any node, the cost of the path from this node to the goal is estimated,then the total cost of a path from the start through the current node to the goal can be estimated
- The estimated cost of the solution path = the cost of the path from the start to the present node + the estimated cost of the remaining path to the goal
- Select the path with the smallest such estimate to expand next
- The estimated total cost is $f(n) = g(n) + h(n)$

Branch and bound with underestimates: Procedure

- To conduct a branch and bound search with dynamic programming:

- 1. Form a queue of partial paths. Let the initial queue consist of the zero-length, zero-step path from the root node to nowhere.
- 2. Until the queue is empty or the goal has been reached, determine if the first path in the queue reaches the goal node.
  - 2a. If the first path does not reach the goal node:
  - 2a1. Remove the first path from the queue.
  - 2a2. Form new paths from the removed path by extending one step.
  - 2a3. Add the new paths to the queue.
  - 2a4. Sort the queue by the *sum of* cost accumulated so far *and a lower-bound estimate of the cost remaining*, with least-cost paths in front.

Avoiding fruitless paths

- Good heuristics can turn bad
  - They are just guesses
  - How do we avoid a deep fruitless path?

  Use a compound heuristic

  Factor in history

  F(n) = g(n) + h(n)

  g(n) – current depth

  h (n) – original heuristic

  Called Algorithm A

The A* search

- • The A* is a branch and bound search with dynamic programming and an underestimate of the remaining cost to the goal
- • To conduct an A* search:

1. Form a queue of partial paths. Let the initial queue consist of the zerolength,zero-step path from the root node to nowhere.
2. Until the queue is empty or the goal has been reached, determine if the first path in the queue reaches the goal node.

- 2a. If the first path does not reach the goal node:
- 2a1. Remove the first path from the queue.
- 2a2. Form new paths from the removed path by extending one step.
- 2a3. Add the new paths to the queue.
- 2a4. Sort the queue by the *sum of* cost accumulated so far *and a*
- *lower-bound estimate of the cost remaining*, with least-cost paths
- in front.
- 2a5. *If two or more paths reach a common node, delete all those*
- *paths except the one that reaches the common node with the*
- *minimum cost.*

The A* search: discussion

- • At any one time, the queue may contain states at different levels of the state space graph, allowing full flexibility in changing focus of the search
- • Since $f(n) = g(n) + h(n)$, the $h(n)$ value guides search toward heuristically promising states while the $g(n)$ value prevents search from persisting indefinitely on a fruitless search
- • The function $h(n)$ relies upon some heuristic information available from the problem domain. It changes from heuristic to heuristic, and from problem to problem
- • The design of good heuristics is a difficult, empirical problem
- • $h(n)$ in general weights how close node n is to the goal
- • In the 8-puzzle problem, if the cost of the path is the total number of tile movements necessary to transform a starting configuration to a required one, then the number of misplaced tiles between node n and the goal can be used as the heuristic.

The evaluation function

- • In searching for the optimal path, one uses the evaluation function $f(n)$, which defines the cost of a path from the start node, through node n, to the goal node
- • Denote $k(ni,nj)$ as the actual cost of a minimal cost path between two arbitrary nodes ni and nj
- • Define an evaluation function $f^*(n) = g^*(n) + h^*(n)$:

- $g^*(n)$ is the actual cost of the shortest path from the start node to node n,i.e., $g^*(n) = k(S,n)$
- $h^*(n)$ is the actual cost of the shortest path from n to the goal,i.e., if G is a set of goals, then $h^*(n)$ = minimum $k(n, nj)$ for all $nj \in G$
- therefore $f^*(n)$ is the actual cost of the optimal path from a start node to a goal node that passes through node n

The evaluation function

- • Since estimates are used, the actual value of the evaluation function, i.e., $f^*(n)$, is not known until the goal has been reached or the graph has been completely searched
- • $f(n) = g(n) + h(n)$ is therefore a close approximation of $f^*(n)$:
- - $g(n)$ is the cost of the minimum-cost path leading to node n uncovered so far, and it is a reasonable estimate of $g^*(n)$
- - note that $g(n) >= g^*(n)$
- - $h^*(n)$ may not be computed but it is possible to determine the heuristic estimate $h(n)$ is bounded,
- • $h(n)$ is called an admissible heuristic if it is bounded by $h^*(n)$

Admissibility and informedness

- Theorem:
- The A* is admissible if an admissible heuristic is used.

(That is, A* always find a minimal cost path to a solution whenever such a path exists.)

- Definition

For two A* heuristics *h*1 and *h*2, if *h*1(*n*)< *h*2(*n*) for all states *n* in the search space, heuristic *h*2 is said to be more informed than *h*1, or *h*2 dominates *h*1.

- Application:
- A more informed strategy (i.e., one that has a heuristic value which is closer to the actual remaining cost) would expand fewer nodes. It is always better to use a heuristic function with higher values, as long as it does not overestimate.

Complexity issues

- The more informed a heuristic, the less space the A* needs to expand to get the optimal solution.
- The computational complexity of an A* search strategy depends on two parameters:
  - The computational complexity of evaluating the heuristic $h(n)$.
  - The number of nodes which were expanded by A*.
- The more informed a strategy is (i.e., the closer $h(n)$ is to $h^*(n)$) the less nodes the search expands. On the other hand, estimating accurately (i.e., $h(n)$ close to $h^*(n)$) may prove computationally expensive.
- One must be careful that the computations necessary to employ the more informed heuristic are not so inefficient as to offset the gains from reducing the number of states searched.
- In certain instances, one may use a heuristic that is not strictly an underestimate. By doing so, one does not guarantee the admissibility of the search, but the search may terminate quickly with or near optimal path.

The End!