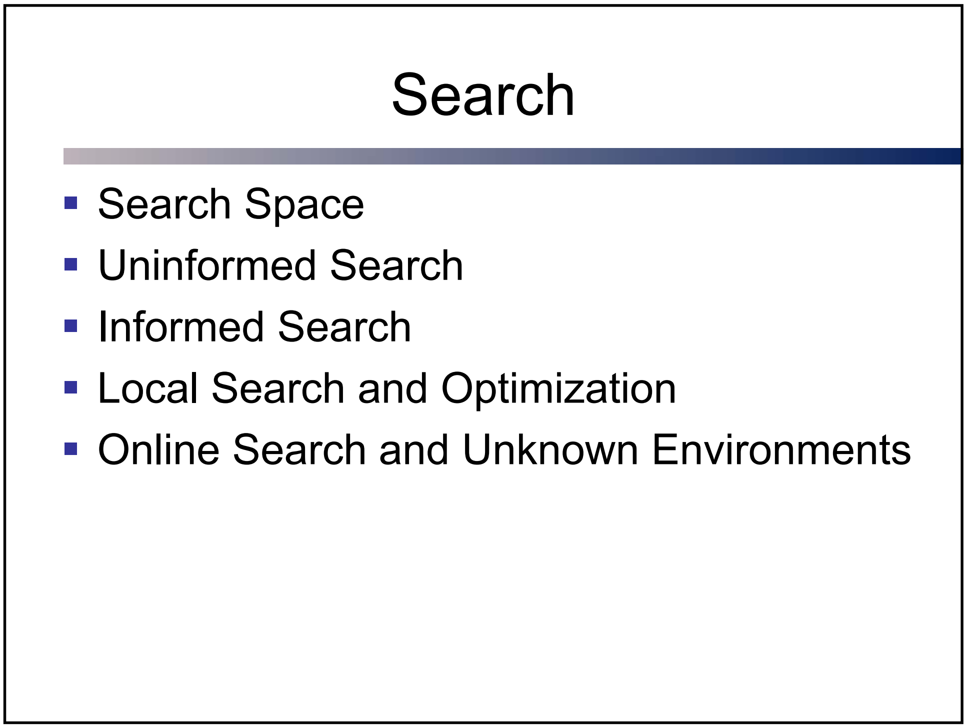
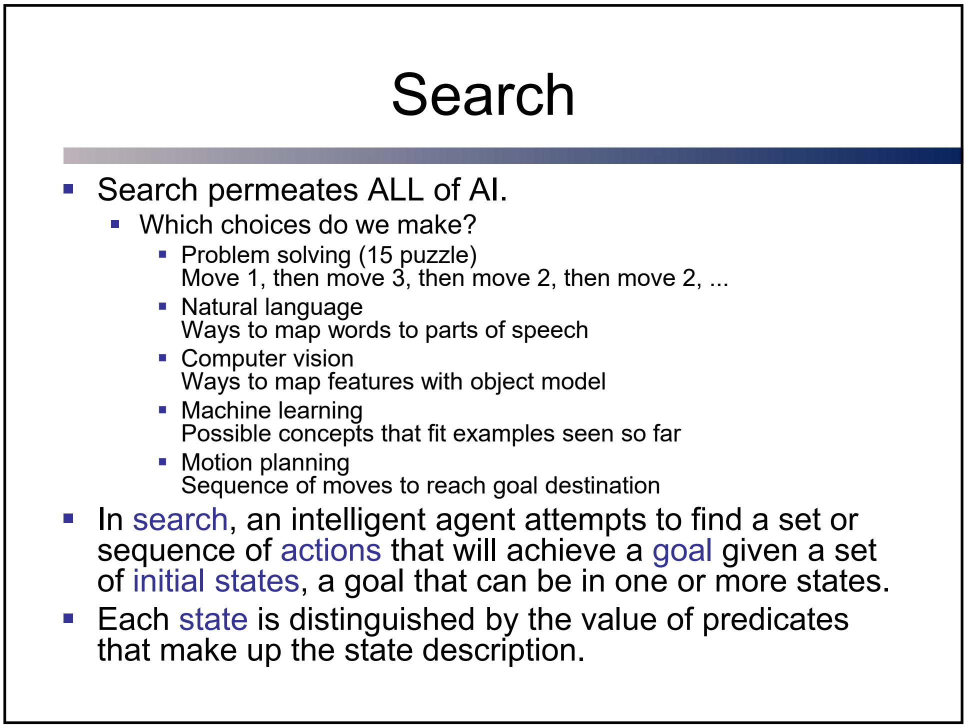


Search

- Search Space
- Uninformed Search
- Informed Search
- Local Search and Optimization
- Online Search and Unknown Environments



Search

- Search permeates ALL of AI.
  - Which choices do we make?
    - Problem solving (15 puzzle)
      Move 1, then move 3, then move 2, then move 2, ...
    - Natural language
      Ways to map words to parts of speech
    - Computer vision
      Ways to map features with object model
    - Machine learning
      Possible concepts that fit examples seen so far
    - Motion planning
      Sequence of moves to reach goal destination
- In search, an intelligent agent attempts to find a set or sequence of actions that will achieve a goal given a set of initial states, a goal that can be in one or more states.
- Each state is distinguished by the value of predicates that make up the state description.

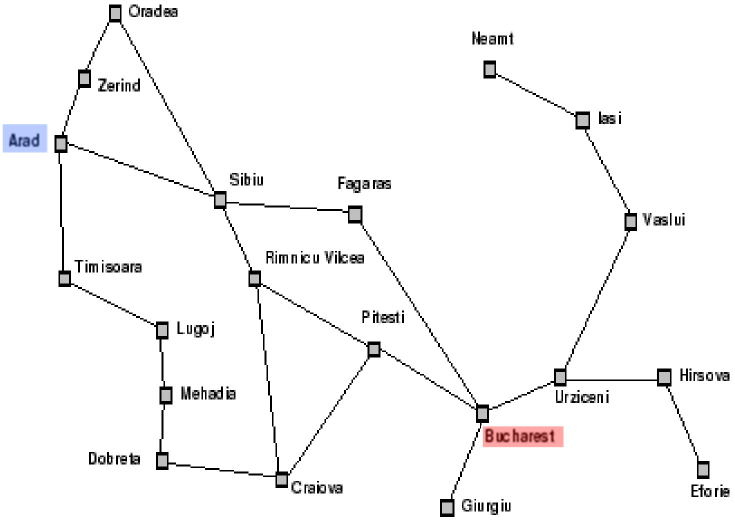
Problem-Solving Agent

```
SimpleProblemSolvingAgent(percept)
   state = UpdateState(state, percept)
   if sequence is empty then
      goal = FormulateGoal(state)
      problem = FormulateProblem(state, g)
      sequence = Search(problem)
      if sequence = FAIL the return NULL
   action = First(sequence)
   sequence = Rest(sequence)
   return action
```

Example

- On holiday in Romania, currently in Arad. Must reach Bucharest tomorrow.
  - **Formulate goal:** Be in Bucharest
  - **Formulate problem:** states are cities, operators are to drive between the pairs of cities
  - **Find solution:** find the sequence of cities (e.g., Arad, Sibiu, Fagaras, Bucharest) that leads from current state to a state meeting the goal condition

Search Space



Search Space Terminology

- World State (state)
A description of a possible state of the world (includes all features of the world that are pertinent to the problem)
- Initial State
A description of all pertinent aspects of the world state in which the agent starts the search
- Goal (Goal Condition)
Conditions the agent is trying to meet
(Have $1,000,000)
- Goal State
Any world state which meets the goal conditions
(Thursday, have $1,000,000, live in NYC)
(Friday, have $1,000,000, live in Valparaiso)
- Action
Function that transitions from one state to another

Search Space Terminology (2)

- Problem Formulation
  - Describe a general problem as a search problem
- Solution
  - Sequence of actions that transitions the agent from the initial state to a goal state
- Solution Cost (additive)
  - Sum of distances, number of operators, cost of operators, etc.
- Search
  - The process of looking for a solution
- Search algorithm - algorithm that takes problem as input and returns solution
  - We are searching through a space of possible world states
- Execution
  - Process of executing sequence of actions that comprises problem solution

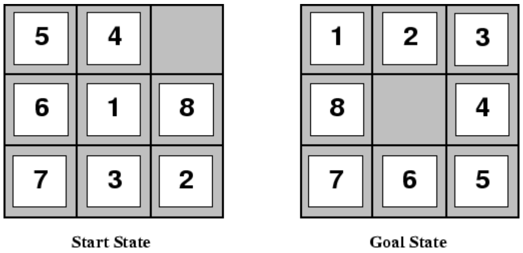
Agent's Process

- 3 main steps
  1. Formulate
  2. Search
  3. Execute

Problem Formulation

- A single-state search problem is defined by the
  - Initial state (e.g., Arad)
  - Operators (Arad → Zerind, Arad → Sibiu, etc.)
  - Goal test (e.g., at Bucharest)
  - Solution cost (path cost)

Eight Puzzle Problem



5	4	
6	1	8
7	3	2

Start State

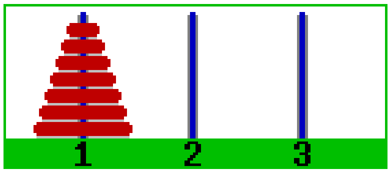
1	2	3
8		4
7	6	5

Goal State

- What is the problem formulation?
- States:
- Actions:
- Goal test:
- Path cost:

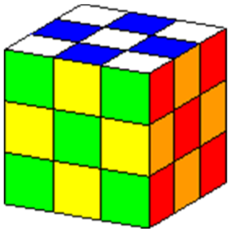
Towers of Hanoi Problem

- States: combinations of poles and disks
- Operators: move disk x from pole y to pole z subject to constraints
- Goal test: disks from smallest to largest on goal pole
- Path cost: 1 per move



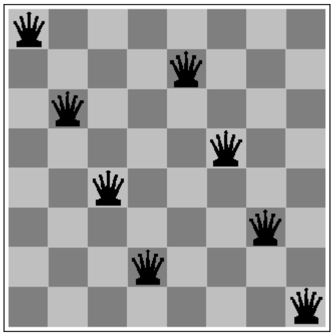
Rubik’s Cube

- States: list of colors for each cell on each face
- Operators: rotate row x or column y on face z
- Goal: Each face has all one color
- Path Cost: 1 per twist
- http://www.math.umass.edu/~mreid/Rubik/optimal_solver.html



Eight Queens Problem

- States: locations of 8 queens on chess board
- Operators: move queen x to row y and column z
- Goal: no queen can attack another
- Path Cost: 1 per move



Sample Search Problems

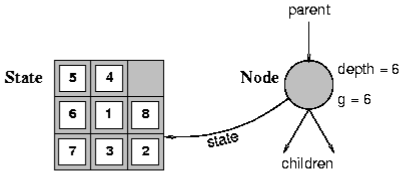
- Graph coloring problem
- Protein folding problem
- Game playing
- Airline travel
- Proving algebraic equalities
- Robot motion planning

Agent’s Process

- 3 main steps
  1. Formulate
  2. Search
  3. Execute

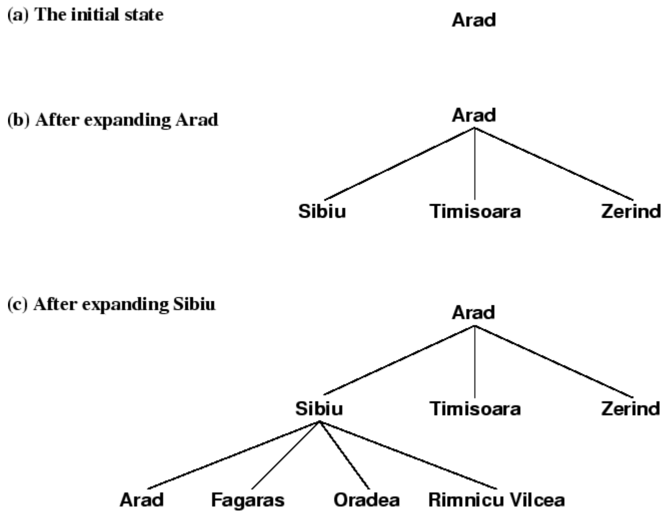
View Search Space as a Tree

- States are nodes
- Actions are arcs
- Initial state is root



- Solution is path from root to goal node
- Arcs sometimes have associated costs
- Possible resulting states are children of a node

General Search Example



(a) The initial state

Arad

(b) After expanding Arad

Arad

Sibiu Timisoara Zerind

(c) After expanding Sibiu

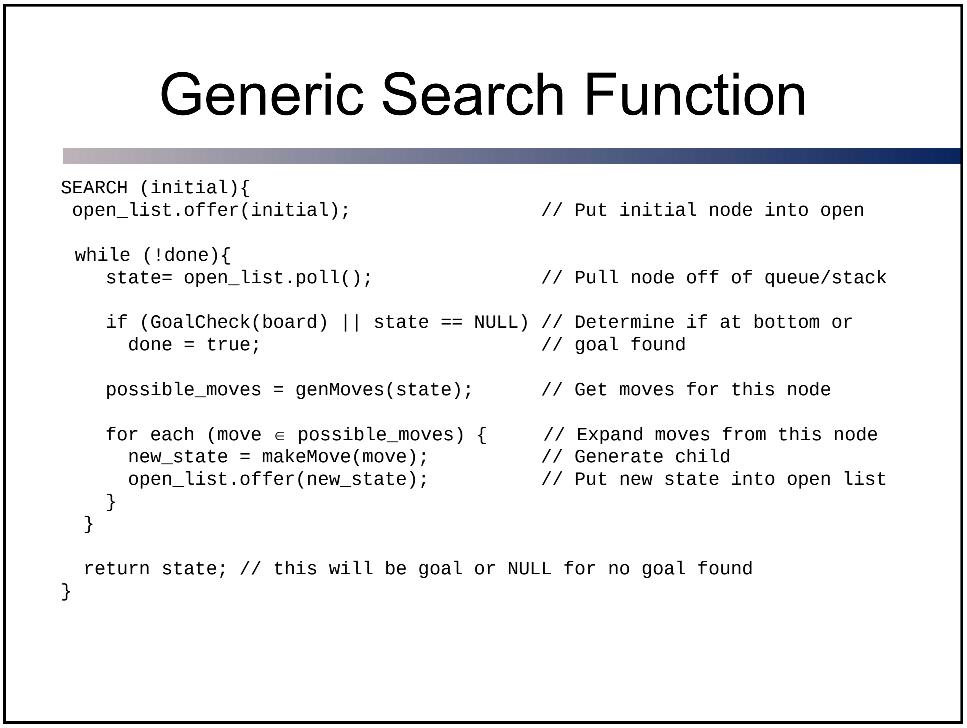
Arad

Sibiu Timisoara Zerind

Arad Fagaras Oradea Rimnicu Vilcea

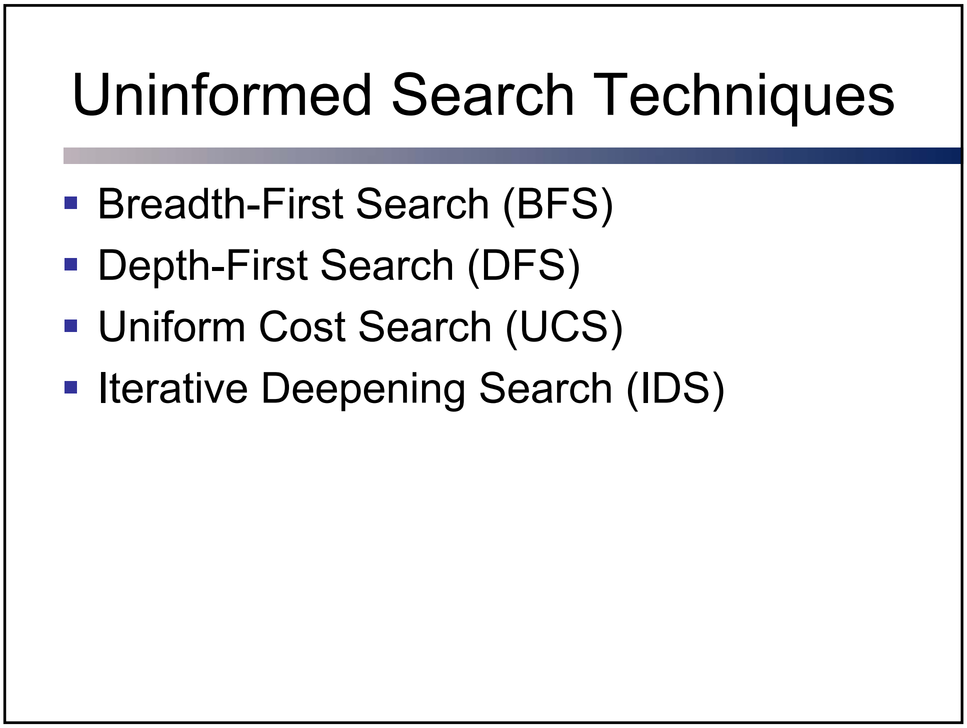
Search Strategies

- Search strategies differ in queuing-function portion of algorithm
- Performance issues to keep in mind
  - Completeness (always find solution)
  - Cost of search (time and space)
  - Cost of solution, optimal solution
  - Make use of knowledge of the domain “blind search” vs. “informed search”



Generic Search Function

```
SEARCH (initial){
 open_list.offer(initial);                 // Put initial node into open

 while (!done){
    state= open_list.poll();               // Pull node off of queue/stack

    if (GoalCheck(board) || state == NULL) // Determine if at bottom or
      done = true;                         // goal found

    possible_moves = genMoves(state);      // Get moves for this node

    for each (move ∈ possible_moves) {     // Expand moves from this node
      new_state = makeMove(move);          // Generate child
      open_list.offer(new_state);          // Put new state into open list
    }
  }

  return state; // this will be goal or NULL for no goal found
}
```



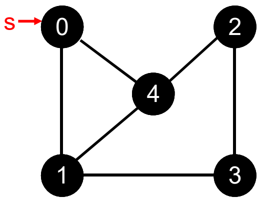
Uninformed Search Techniques

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Uniform Cost Search (UCS)
- Iterative Deepening Search (IDS)

Breadth-First Search

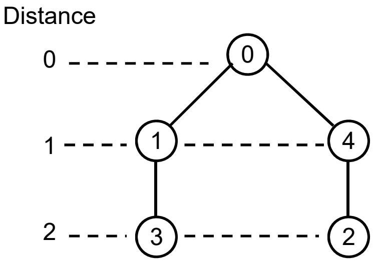
- Queuing-function is enqueue-at-end
- Generate all children of a state, add the children to the end of the queue
    - Net effect is all nodes at one level are expanded before any nodes at the next level
- Level-by-level search
- Order in which children are inserted is arbitrary (4 children, which is first?)
- In tree, assume children are considered left-to-right unless ordered
- Number of children is “branching factor” b

BFS Example



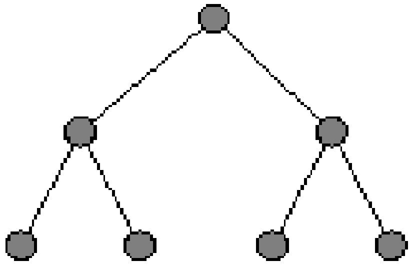
Q = {[illegible]}

Adj([illegible]) = [illegible]

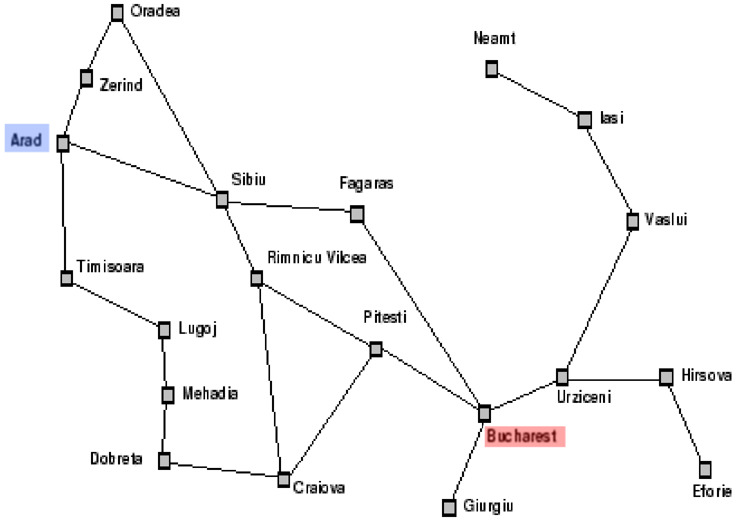


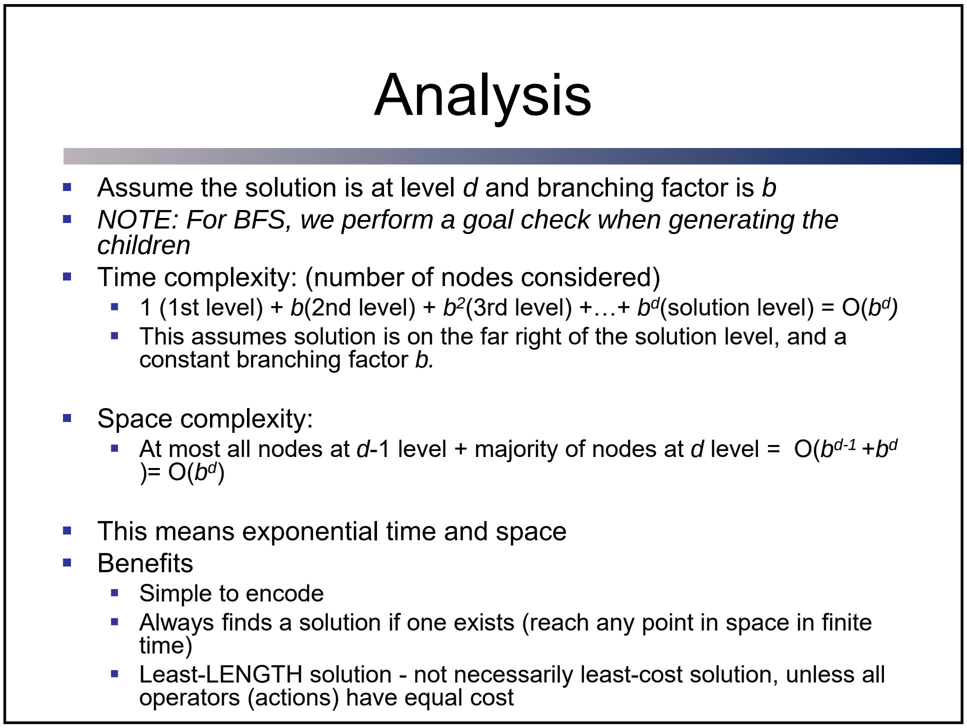
Examples

- Breadth-first expansion of search tree with a branching factor of 2



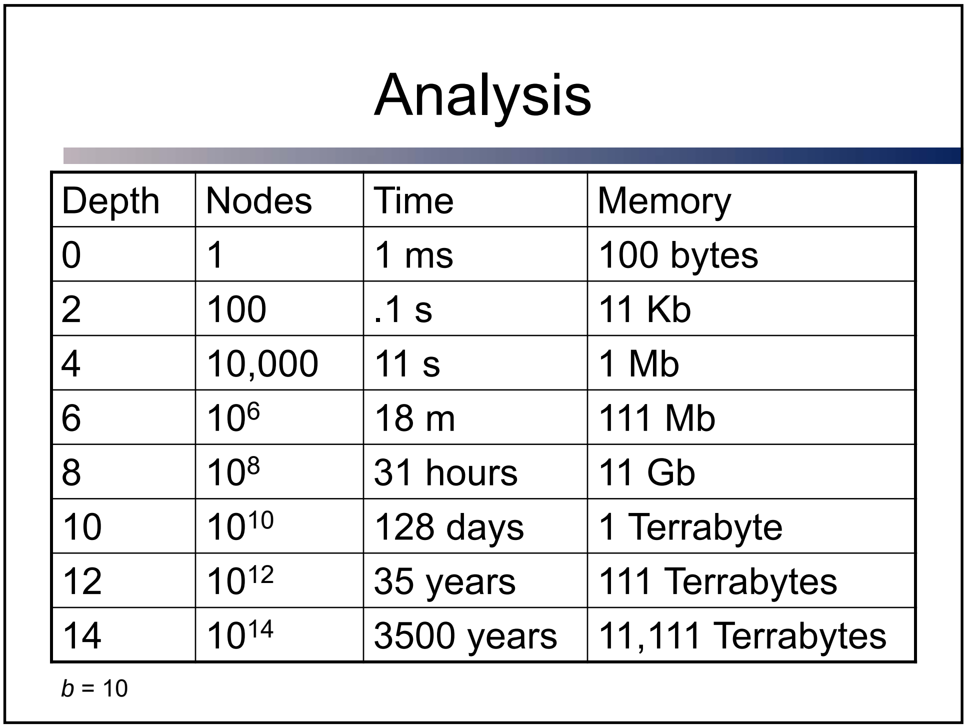
Search Space





Analysis

- Assume the solution is at level *d* and branching factor is *b*
- *NOTE: For BFS, we perform a goal check when generating the children*
- Time complexity: (number of nodes considered)
  - 1 (1st level) + *b*(2nd level) + b^2(3rd level) +…+ b^d(solution level) = O(b^d)
  - This assumes solution is on the far right of the solution level, and a constant branching factor *b.*
- Space complexity:
  - At most all nodes at *d*-1 level + majority of nodes at *d* level = O(b^{d-1} +b^d)= O(b^d)
- This means exponential time and space
- Benefits
  - Simple to encode
  - Always finds a solution if one exists (reach any point in space in finite time)
  - Least-LENGTH solution - not necessarily least-cost solution, unless all operators (actions) have equal cost



Analysis

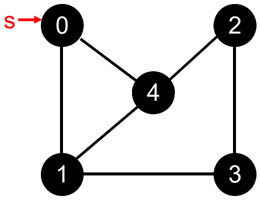
Depth	Nodes	Time	Memory
0	1	1 ms	100 bytes
2	100	.1 s	11 Kb
4	10,000	11 s	1 Mb
6	10^6	18 m	111 Mb
8	10^8	31 hours	11 Gb
10	10^{10}	128 days	1 Terrabyte
12	10^{12}	35 years	111 Terrabytes
14	10^{14}	3500 years	11,111 Terrabytes

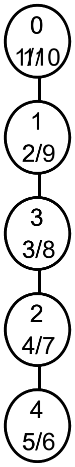
b = 10

Depth-First Search

- Queueing-function is enqueue-at-front
- BFS uses FIFO queue, DFS uses LIFO stack
- Net effect is to follow leftmost path to the bottom, then incrementally backtrack

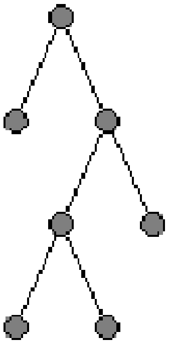
DFS Example



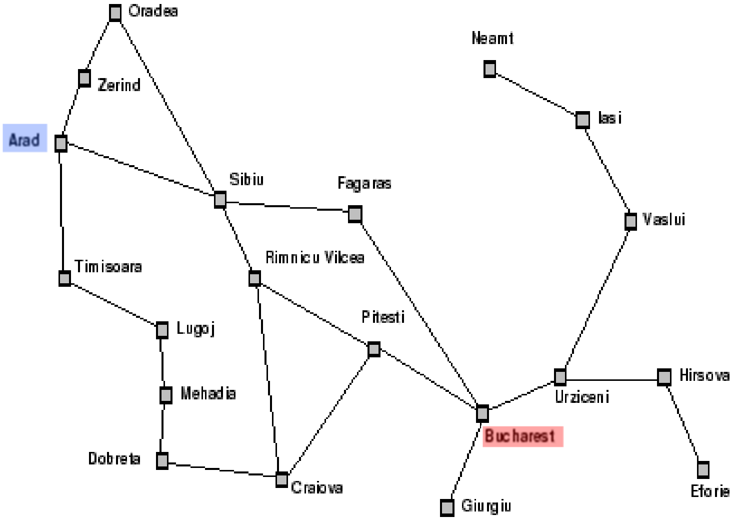


Examples

- Depth-first expansion of search tree with a branching factor of 2



Search Space

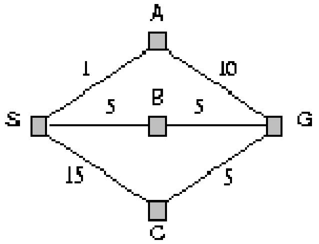


Analysis

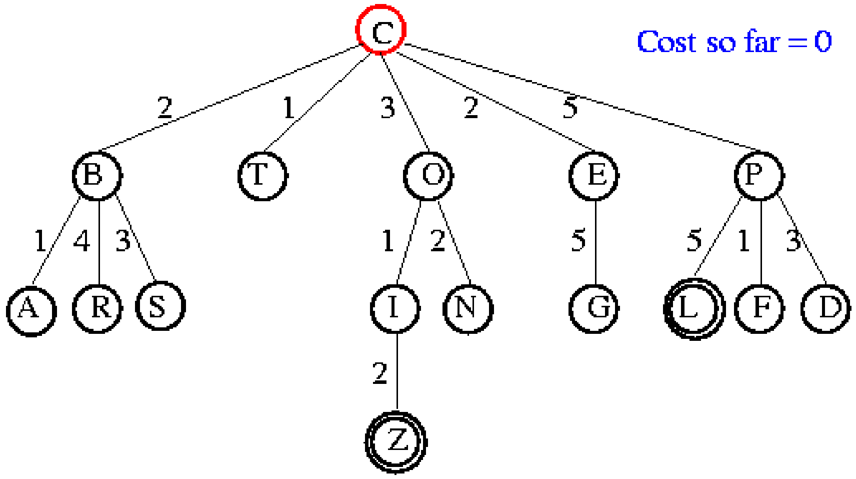
- Time complexity:
  - In the worst case, have to search entire search space
    Solution may be at level *d*, but tree may go to level *m*, *m* ≥ *d*, so run time is O(b^m)
  - Particularly bad if tree is infinitely deep
- Space complexity:
  - Only have to save 1 set of children at each level
    1+*b*+*b*+…+*b*(*m* levels total) = O(*bm*)
- May not always find solution
- Solution found is not always least-length or least-cost
- Benefits
  - If many solutions, can find one quickly (quickly moves to depth d)
  - Simple to implement
  - Space usually bigger constraint than time, so more usable than BFS for large problems

Uniform Cost Search (Branch and Bound Search)

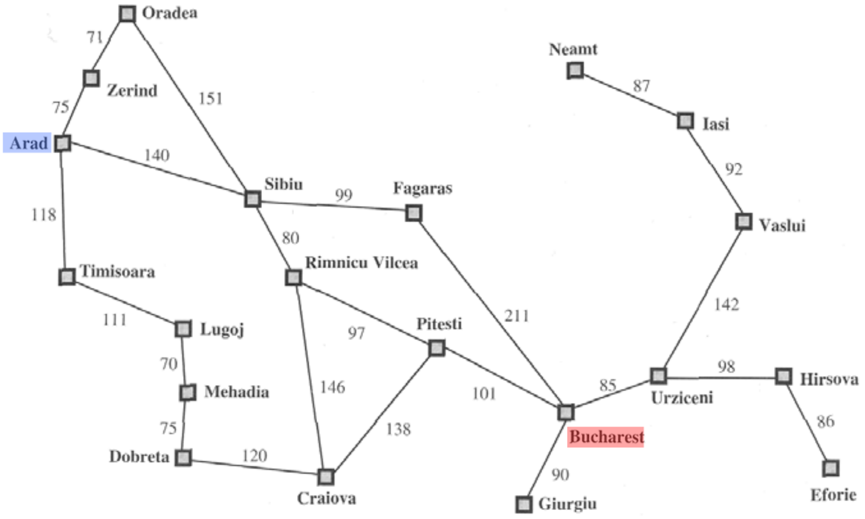
- Queueing-function is sort-by-cost-so-far or sort-by-*g*
- Cost from root node to current node *n* is *g*(*n*), which adds up individual action costs along path from root to *n*
- Because cheapest path length is always picked until solution is reached, first solution found is least-cost (optimal) solution
- Space and time complexities can be as much as $O(b^m)$, depending on the nature of the plan costs $\theta\left(b^{1+\lfloor C^*/\epsilon \rfloor}\right)$ where $C*$ is the optimal cost and ϵ is the average action cost.
- If $\epsilon = 1$ then we have BFS, except with the goal check on pop end with $O\left(b^{d+1}\right)$



UCS Example



Search Space

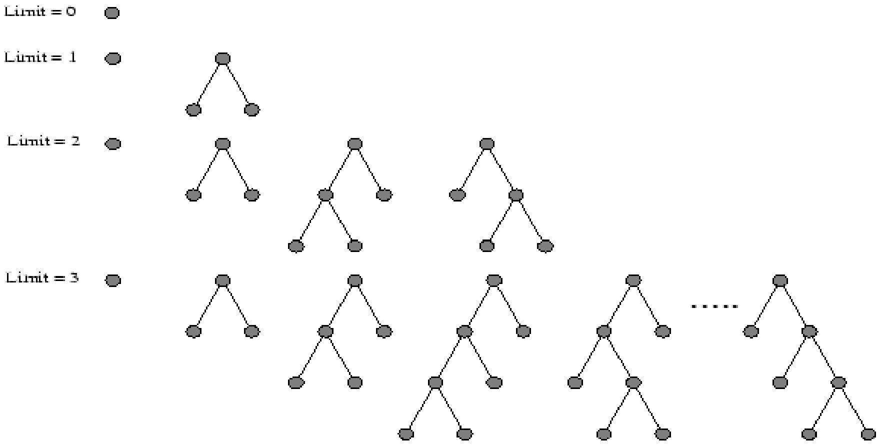


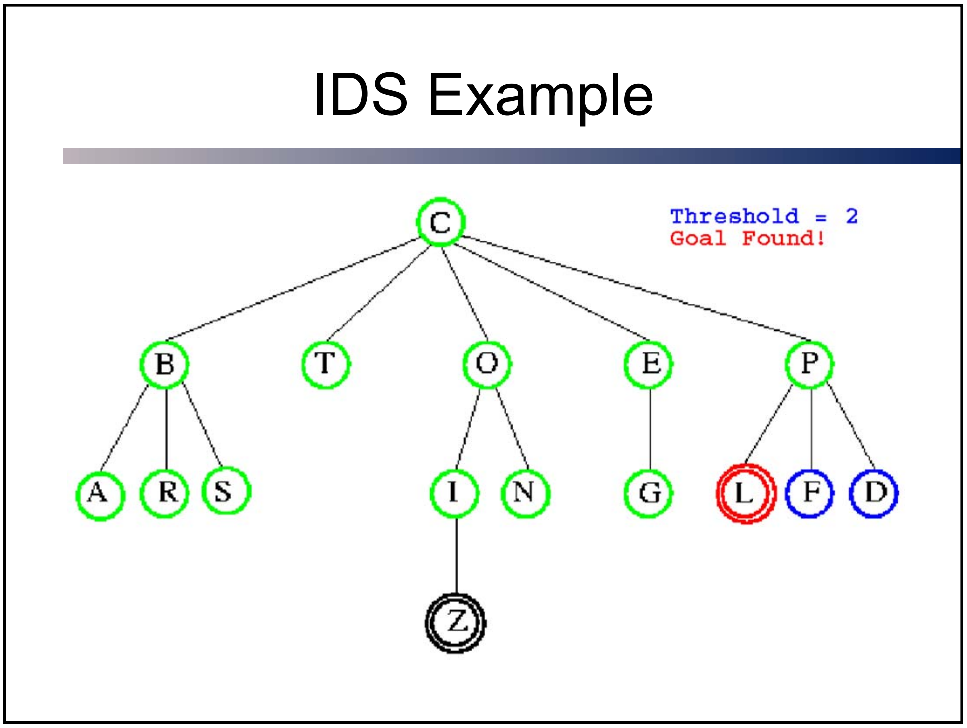
Iterative Deepening Search (IDS)

- DFS with a cutoff bound
- Usually backtrack when cannot add children
  - no children to add to front of queue, no next state on queue to pop

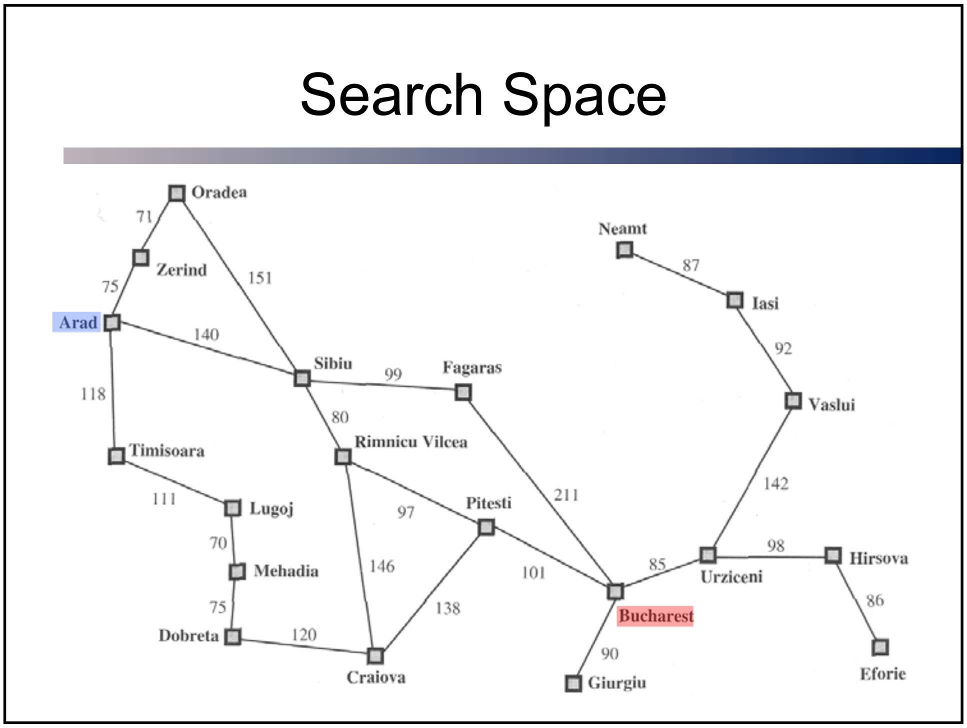
- Queueing-function is enqueue-at-front
- As before, but (expanded state) ONLY returns children with depth(children) ≤ threshold
- This prevents DFS from going down infinite path However, we many not find solution!
- Try one threshold - if no solution, increase threshold and start again from root

Example





IDS Example



Search Space

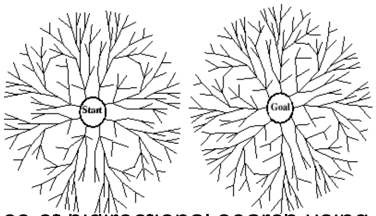
Analysis

- Repeat some work, but repeated work is only fraction of work on last (unrepeated) level

  [1]+[1+b]+[1+b+b^2]+…+[1+b+b^2+…+b^d]
- Repeated work is approximately 1/b of total work (negligible)
- Is there a better way to decide thresholds? Yes!
  - An informed search like IDA*

Bidirectional Search

- Search forward from initial state to goal AND backward from goal to initial state
- Prune much of the search space
- Must consider:
  - Which goal state to use (set of goal states)?
  - How to determine when two searches overlap?
  - Which search to use for each direction (DFS probably not good choice)
    - Figure shows two breadth-first searches
    - Backwards search can be difficulty for some problems



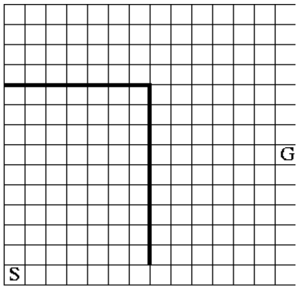
- Run time and space of bidirectional search using an uninformed searches is O($b^{d/2}$).

Avoid Repeating States

- Speed search by avoiding state repetition
- Do not return to the parent state
  - e.g., in 8 puzzle problem, do not allow the Up move right after a Down move
- Do not create solution paths with cycles
- Do not generate any repeated states
  - need to store and check a potentially large number of states

Operator Ordering

- Imagine states are cells in a maze, and you can move N, E, S, or W.
- What would BFS do, assuming it always expanded the E child first, then N, then W, then S?
- What about DFS?
- What if the order changed to N, E, S, W, and loops are prevented?



Informed Search

- Besides not repeating states and operator ordering how else can we speed search?
- Use information in and about the domain to guide our search strategy.
  - Distance from the start
    - As time, cost, fuel, etc
  - Distance to the goal
    - If I go ‘left’ will I reach the goal faster than if I go ‘right’?

Informed Search Techniques

- Best-First Search
- Hill Climbing
- Beam Search
- A*
- IDA*
- SMA*
- Many others….

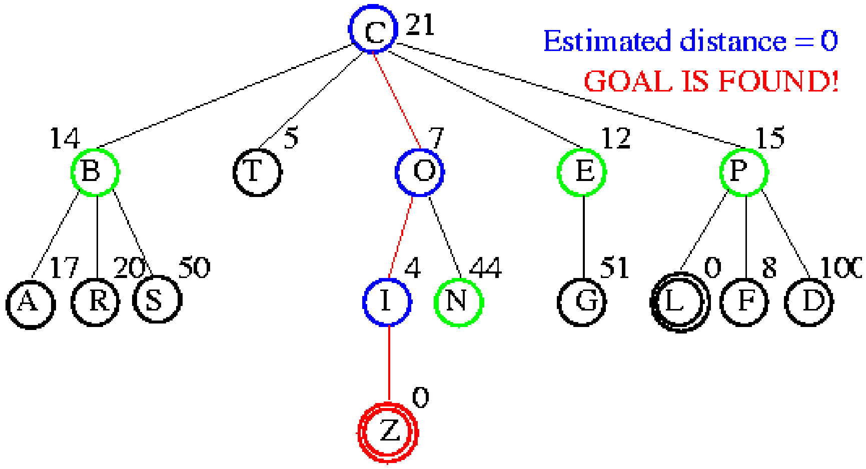
Informed Searches

- We'll learn related terms such as heuristics, optimal solution, informedness, hill climbing problems (local maxima/minima, plateau, foothill problem) and admissibility.
- *g*(*n*) = estimated cost from initial state to node n
  - Uniform cost search (branch-and-bound search) uses this measure.
- *h*(*n*) = estimated (guessed) distance from node n to closest goal
  - The function h is our HEURISTIC.
- *f*(*n*) = *g*(*n*) + *h*(*n*)
  - The combination provides even more information as seen in A*
- Methods which use h to guide search are called heuristic search methods.

Best-First Search

- Queueing-function is sort-by-*h*
  - Similar to UCS but sorting on *h*(*n*) instead of *g*(*n*)
- *h*(*n*) is estimated distance remaining to the nearest goal state
- Best-First Search is only as good as its heuristic
- Example Heuristic: Manhattan Distance Function for 8 puzzle

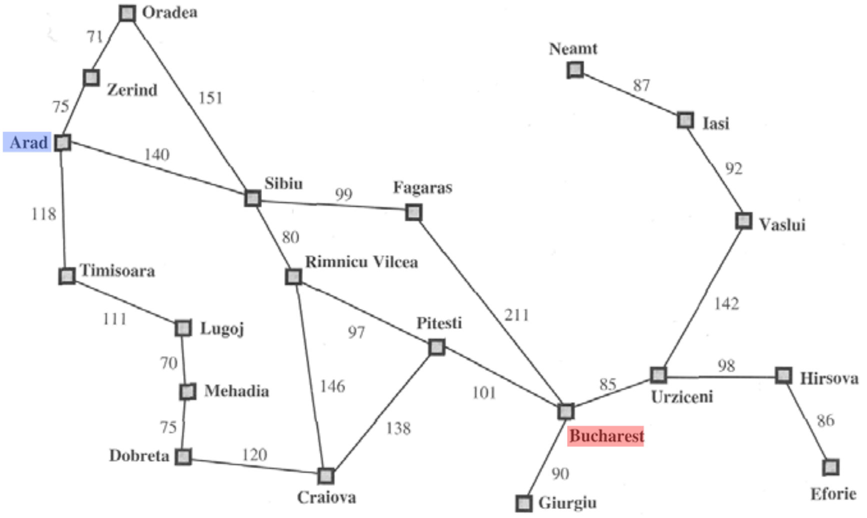
Best-FS Example



Estimated distance = 0

GOAL IS FOUND!

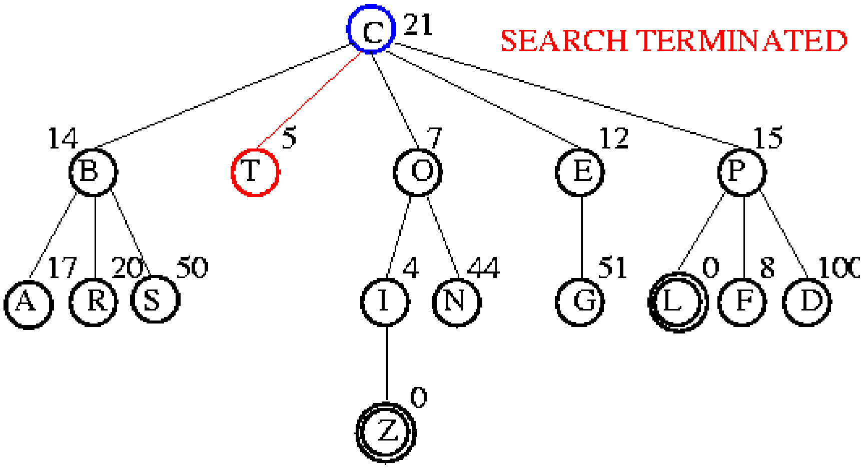
Search Space



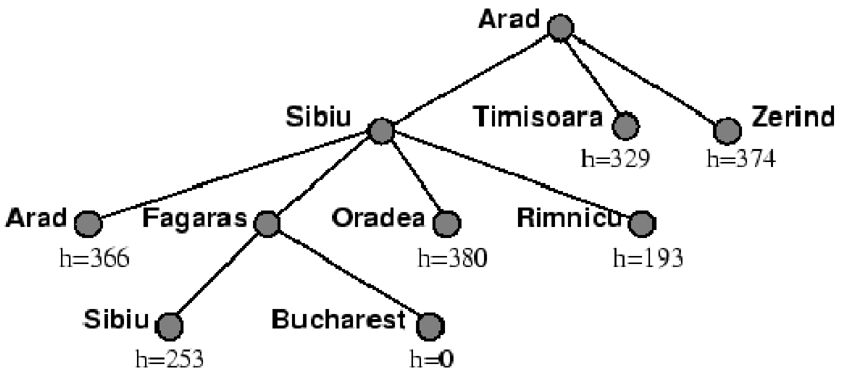
Hill Climbing (Greedy Search)

- Hill Climbing is also guided by *h*, but it saves time and space by only keeping the single best state seen so far.
- queueing-fn is (first (sort-by-*h* open))
- Best-First Search is tentative, Hill Climbing is irrevocable
- Advantage: much faster, takes less memory
  If good heuristic, then Hill Climbing will lead immediately to the goal
- Disadvantage: if bad heuristic, may prune away all the goals
  - Gets stuck in the tree above

Hill Climbing Example



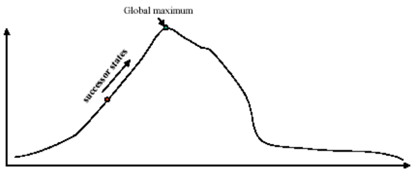
Example



Hill Climbing Issues

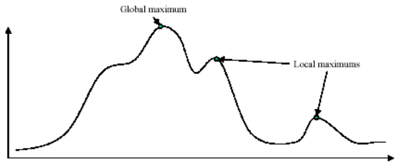
- Also referred to as gradient descent (an iterative improvement algorithm)

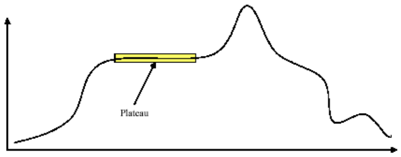
- In optimization problems, the *path* is irrelevant, the goal state is the solution
- Start with a configuration (often a solution) and attempt to improve its quality
  - state space becomes set of "complete" configurations
  - goal is to find the *optimal* solution



Hill Climbing Issues

- Foothill problem / local maxima / local minima
- Plateau problem
- Both can be solved with random walk or more steps
  - We will revisit this when we discuss local search

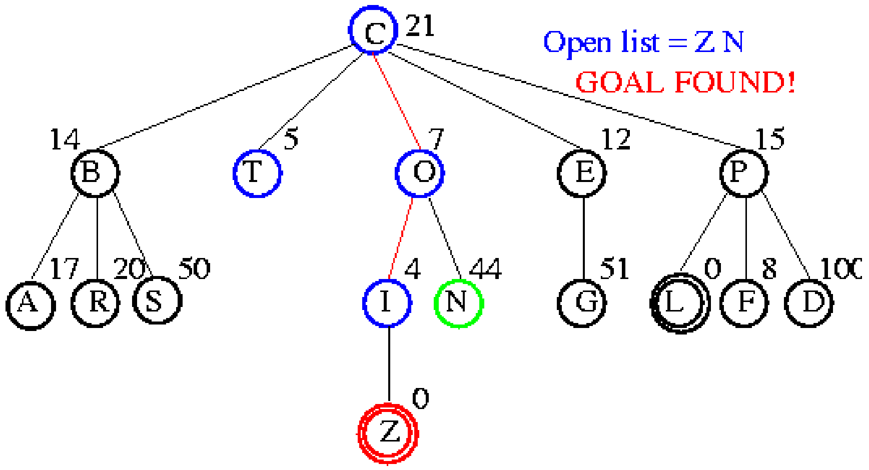




Beam Search

- Keep *n* best alternatives.
- open_list = (first-*n* (sort-by-*h* (expand state)))
- *n* is the “beam width”
  - *n* = 1, Hill Climbing
  - *n* = infinity, Best-First Search

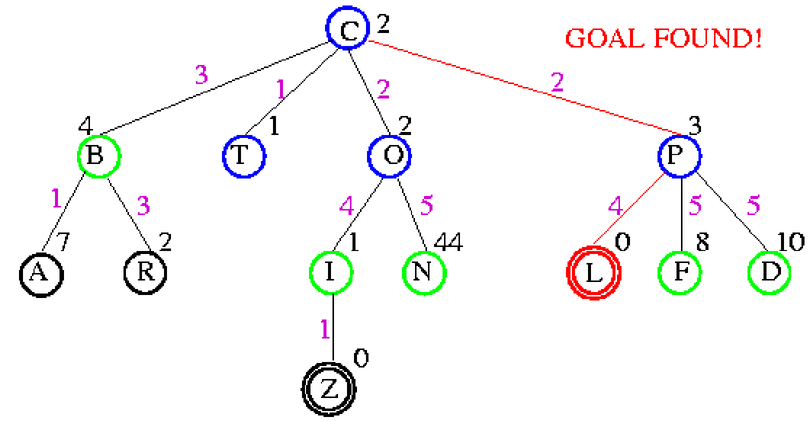
Beam Search Example



A* Search

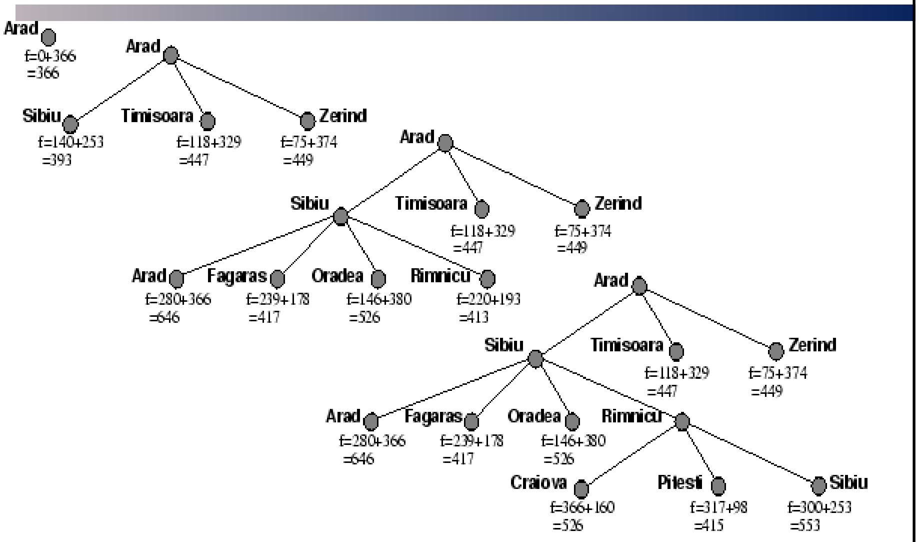
- Note that UCS and best-first search both improve the search
- UCS (B&B) makes sure that the solution path is low cost, and Best-First helps to find a solution quickly
- A* combines these approaches, using the evaluation function
- *f*(*n*) = *g*(*n*) + *h*(*n*)
- *g*(*n*) - cost so far to reach n from initial state
  h(*n*) - estimated cost to nearest goal state from n
  f(*n*) - estimated total cost of path through n to goal
- The search is guided by the function *f* the open list is sort-by-*f*
- If a heuristic function is wrong (bad estimate), it either overestimates (guesses too high) or underestimates (guesses too low).
- For A*, overestimating is much worse than underestimating.

A* Search Example



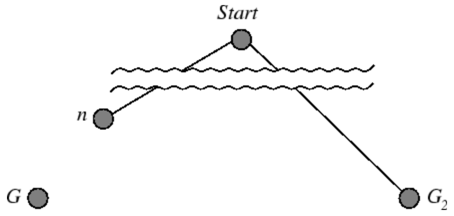
Open List = L (6+0=6), B (3+4=7)
I (6+1=7), F (7+8=15), D (7+10=17), N (7+44=51)

Example



Optimality of A*

- Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1.



$f(G_2) = g(G_2)$ since $h(G_2)=0$

$> g(G_1)$ since G_2 is suboptimal

$\geq f(n)$ since h is admissable

- Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Suboptimal A*

- The heuristic function in A* is:

$$f(n) = g(n) + h(n)$$

- Instead where n is the current node, use the following formula

$$f(n) = g(n) + ((1 + e^{w(n)})\ h(n))$$

where e (epsilon) is a constant between 0 and 1, and $w(n)$ is

$w(n) = h(n)/h(s)$, if $h(n) \leq h(s)$

1, otherwise

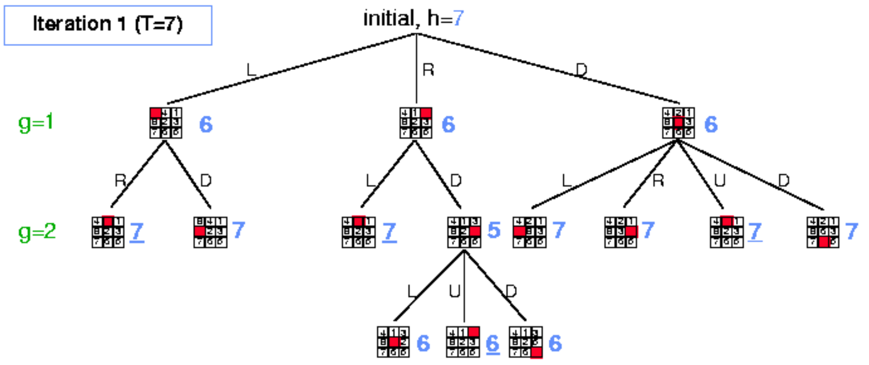
where $h(s)$ is the initial estimate of the path from the start node s to the target node.

- Can control quality of the solution by decreasing epsilon. As epsilon gets larger the quality decays.

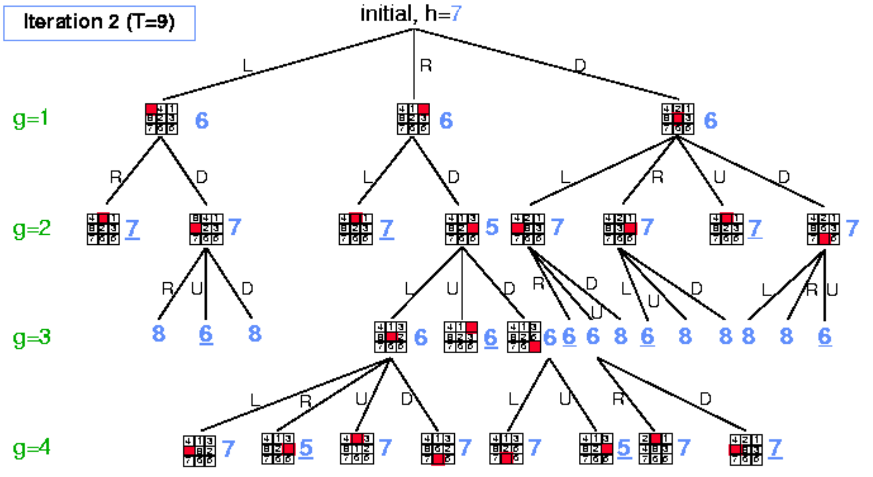
IDA* Korf, 1984

- Series of Depth-First Searches
  - Benefits from linear space requirement of DFS
- Place cutoff limit
  - Prevents infinite search down a particular branch
- Using A* guide to determine cost limit
  - Ensures optimal solution
  - queueing-fn is enqueue-at-front
  - expand (state) ONLY returns children if *f*(child) ≤ threshold
- Just like iterative deepening search, except cost threshold instead of depth threshold
- Why? Guarantees optimal solution
- Threshold is initially h(root)
- Next threshold is f(min_child), where min_child is cutoff child with minimum f value
- This conservative increase ensures cannot look past optimal cost solution

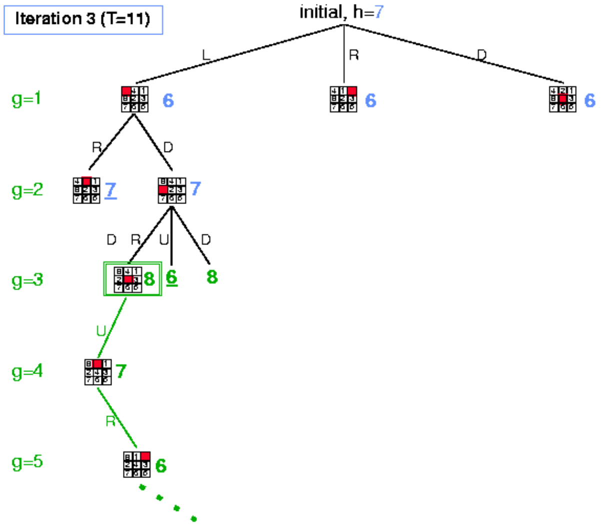
Eight Puzzle Example



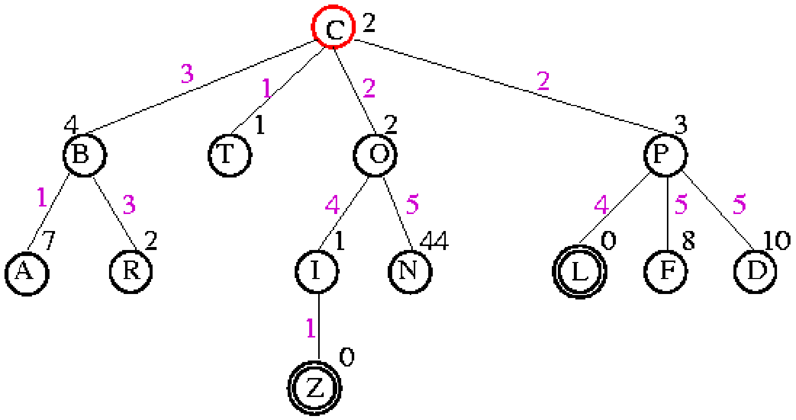
Eight Puzzle Example



Eight Puzzle Example



IDA* Example



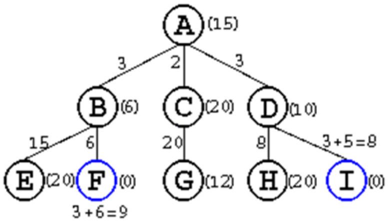
limit = f(C) = 2

Analysis

- Some redundant search, but small amount compared to work done on last iteration
- Dangerous if *f* values are very close
  - If threshold = 21.1 and next value is 21.2, probably only include 1 new node each iteration
- Time: O(b^m)   Space: O(m)
- SMA* search can be used to remember some nodes from one iteration to the next.

Heuristic Functions

- If a heuristic function is wrong (bad estimate), it either overestimates (guesses too high) or underestimates (guesses too low).
- For A*, overestimating is much worse than underestimating.



- Solution cost: ABF = 9 ADI = 8
- Q: Given that we are always going to use heuristic functions that never overestimate the distance to the goal, what type of heuristic function perform best?
  - Those that produce higher *h* values.

Reasons

- A higher *h* value means it is closer to the actual distance
- Any node *n* on the *q* with *f*(*n*) < *f**(*s*) will eventually be selected for expansion by A*.
- This means if a LOT of nodes have a low underestimate, lower than the actual optimal cost, ALL of them will be expanded, resulting in increased search time and space.

Informedness

- If *h1* and *h2* never overestimate distance to the closest goal, and
- For all *x*, *h1*(*x*) > *h2*(*x*), then *h1* “dominates” *h2* *h1* is “more informed” than *h2*
- For example, two heuristics for robot motion planning are:

  h1(*x*): $|x_{goal} - x|$

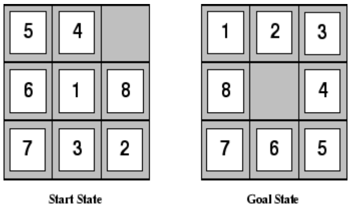
  h2(*x*): Euclidean distance

  $$\sqrt{(x_{goal} - x)^2 + (y_{goal} - y)^2}$$

  Thus *h2* dominates *h1*

Heuristics

- Which of the following heuristics are admissible for the eight puzzle problem?



5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

- *h*(*n*) = Number of tiles in wrong position in state *n*
  - *h*(*goal*) = 0
- *h*(*n*) = Sum of Manhattan distance between each tile and goal location for the tile
  - *h*(*goal*) = 0
- *h1*(*S*) = 7, *h2*(*S*) = 2+3+3+2+4+2+0+2 = 18

Inventing Informed Heuristics

- Admissible heuristics can often be derived from an *exact* solution to a *relaxed* version of the problem
  - a *relaxed* problem is a problem with fewer restrictions
  - e.g., in the 8-puzzle, tiles may only move to the square with a blank, but we can relax this restriction to:
    - a tile can move *anywhere*
    - a tile can move to *any adjacent square*
  - the exact solutions to these two problems are *h1* & *h2* respectively
  - generally, the problem with the fewest (or least) relaxations gives a better heuristic

Typical Search Costs

- Typical search costs with d = 14:
  - IDS = 3,473,941 nodes
  - A*(*h1)*= 539
  - A*(*h2*)= 113

- Typical search costs with d = 24:
  - IDS = too many
  - A*(*h1)*= 39,135
  - A*(*h2*)= 1,641

Admissible Search Algorithms

- An algorithm is *admissible* if, for any graph, it always terminates in an optimal (least-cost) path from a node *s* to a goal node whenever a path from *s* to a goal node exists.
- A* is admissible if
  - All operators costs are > 0
  - All heuristic estimates are ≥ 0, and
  - The heuristic function never overestimates

Informed Search

- More on Heuristics:
  - http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Comparison of Search Techniques

- Let *b* = branching factor
  m = depth of search space
  d = length of solution

	DFS	BFS	UCS	IDS	HC	Beam	A*	IDA*
Complete?	N	Y	Y	Y	N	N	Y	Y
Optimal?	N	N	Y	N	N	N	Y	Y
Heuristic	N	N	N	N	Y	Y	Y	Y
Time	b^m	b^d	$b^{1+\lfloor C^*/\epsilon \rfloor}$	b^d	m	$m*n$	b^m	b^m
Space	m	b^d	b^m	bd	1	n	b^m	bm

Refinements to Search

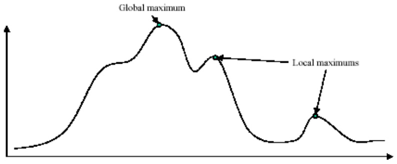
- Local Search
  - Hill Climbing and Local Beam Search
  - Simulated Annealing
  - Genetic Algorithms
  - Evolutionary Computation, Ant Colony Optimization, …
- Online Search and Partially known or unknown environments
  - Real-Time A* (RTA*)
  - Learning Real-Time A* (LRTA*)
  - Lifetime Planning A*,Ant Colony Optimization,…

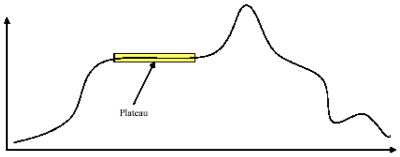
Local Search

- Hill Climbing and Beam Search are local searches
  - As they make an action choice they become dedicated to that choice
- If we <u>don’t</u> require a ‘path’ from the start to the goal. The use of local search provides a quick method for locating a solution
- Pay attention to the optimality and completeness
  - i.e. purely random action is complete and optimal but not efficient.

Hill Climbing Issues

- As mentioned before Hill Climbing and Beam Search suffer from plateau, ridges, and local minima problems
- This means that the solution is not guaranteed to be complete or optimal
- However, our odds can be improved…



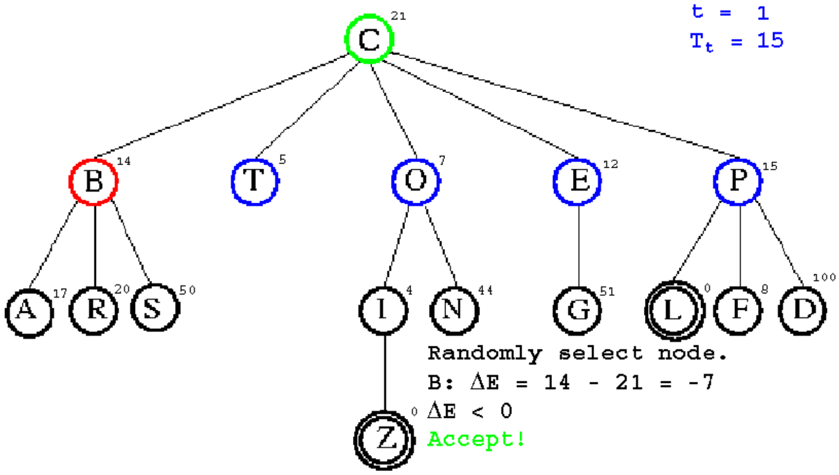


Simulated Annealing

- Idea: Inspired by **annealing** process of gradually cooling a liquid until it freezes
  - escape local maxima by allowing some “bad” moves *but gradually decreasing their size and frequency*
- Implementation
  - A temperature variable, *T*, is used to determine probability of “bad” moves
    - the probability of *T* decreases exponentially with Δ*E*, the badness of the move
    - if *T* is decreased slowly enough ⇒ always reach best state
  - the *schedule* determines the rate at which *T* is lowered
- Very similar to hill climbing, except the **temperature schedule**.
  - As long as the change in energy between the previous state and the current state is less than 0, take the move.
- When temperature is “high”, allow some random moves.
- When temperature “cools”, reduce probability of random move.

SA Example

T_0 = 30 and schedule[*t*] or $T_t = (0.5^t)*T_0$



t = 1
T_t = 15

Randomly select node.
B: ΔE = 14 - 21 = -7
ΔE < 0
Accept!

Simulated Annealing

- Visualization:
  - http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/

Genetic Algorithms (GA)

- A genetic algorithm is an adaptation procedure based on the mechanics of natural genetics and natural selection.
- GAs have 2 essential components:
  1. Survival of the fittest
  2. Recombination
- chromosome = string
- gene = single bit or single subsequence in string, represents 1 attribute

Humans

- DNA made up of 4 nucleic acids (4-bit code)
- 48 chromosomes in humans, each contain 3 billion of these
- $4^{3 \text{ billion}}$ combinations of bits
- Can random search find humans?
  - Assume only 0.1% genome must be discovered, $3(10^6)$ nucleotides
  - Assume parallel search with huge population, 10^{100} humans
  - Assume very short generation, 1 generation / second
  - $3.2(10^{107})$ individuals per year, but $10^{1.8(10^7)}$ alternatives
  - $10^{1.8 10^6}$ years to generate human randomly

- Self-reproduction, self-repair, adaptability are the rule in natural systems, they hardly exist in the artificial world
- Finding and adopting nature's approach to computational design should unlock many doors in science and engineering

GAs Exhibit Search

- Each attempt a GA makes towards a solution is called a chromosome - a sequence of information that can be interpreted as a possible solution
- Typically, a chromosome is represented as a sequence of binary digits. Each digit is a gene
- A GA maintains a collection or population of chromosomes. Each chromosome in the population represents a different guess at the solution.

The GA Procedure

1. Initialize a population (of solution guesses).
2. Do (once for each generation):
   1. Evaluate each chromosome in the population using a fitness function .
   2. Apply GA operators to population to create new population.
3. Finish when solution is reached or number of cycles has reached allowable maximum.

Common Operators

- Reproduction
- Crossover
- Mutation

Example - Mastermind

- Opponent thinks up a five-color sequence:
- R Y B W B
- You make guesses, and the opponent tells you
- 1) How many colors are correct
- 2) How many colors are in right position

R R R B B	3 correct, 2 in position
R Y Y B B	4 correct, 3 in position
R Y B B B	4 correct, 4 in position
R Y B W B	SOLVED!!!

Reproduction

- Select *x* individuals according to their fitness values *f*(*x*) (like beam search)

- Fittest individuals survive (and possibly mate) for next generation

Crossover

- Select two parents
- Select cross site
- Cut and splice pieces of one parent to those of other

```
1 1 | 1 1 1 ⟶ 1 1 | 0 0 0
0 0 | 0 0 0    0 0 | 1 1 1
```

Mutation

- With small probability, randomly alter 1 bit
- Minor operator
- An insurance policy against lost bits, pushes out of local minima

Population:	Goal:
1 1 0 0 0 0	0 1 1 1 1 1
1 0 1 0 0 0	
1 0 0 1 0 0	Mutation needed to find the goal
0 1 0 0 0 0	

Example

- GA codes “guesses” as binary numbers (vector)
- Score is the number of digits that match solution

Solution = 001010

Guess	Score
A) 010101	1
B) 111101	1
C) 011011	3
D) 101100	3

- To “evolve” the correct answer, delete low scorers (A and B) and subject high scorers to “genetic operators”. In particular, produce crossover offspring of C and D at two points.

Example

Crossover	Offspring	Score
C) 01:1011	E) 01:1100	1
D) 10:1100	F) 10:1011	3
C) 0110:11	G) 0110:00	4
D) 1011:00	H) 1011:11	3

Crossover	Offspring	Score
F) 1:01011	I) 1:11000	1
G) 0:11000	J) 0:01011	3
F) 101:011	K) 101:000	4
G) 011:000	L) 011:011	4

Crossover	Offspring	Score	
J) 0010:11	M) 0010:00	5	
K) 1010:00	N) 1010:11	4	
J) 00101:1	O) 00101:0	6	— SOLVED!!!
K) 10100:0	P) 10100:1	3	

Issues with GAs

- How to select original population?
- How to handle non-binary solution types?
- What should the size of the population be?
- What is the optimal mutation rate?
- How are mates picked for crossover?
- How is an individual scored?
- Can any chromosome appear more than once in a population?
- Local minima?
- Multiple goals?
- Parallel algorithms?

GA Additional Operators

- Inversion
  - Invert selected subsequence

  1 0 | 1 1 0 | 1 1 → 1 0 0 1 1 1 1

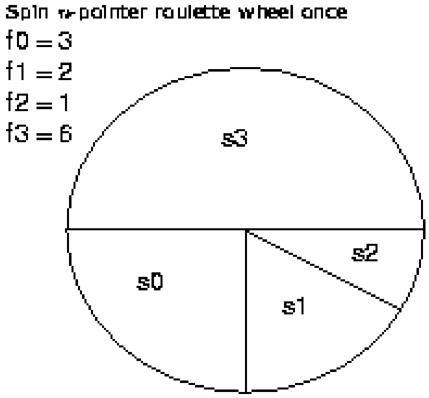
- K-point Crossover
  - Pick k random splice points to crossover parents.

  k=3

  1 1 | 1 1 1 | 1 1 | 1 1 1 1 1 ⟶ 1 1 0 0 0 1 1 0 0 0 0 0
  0 0 | 0 0 0 | 0 0 | 0 0 0 0 0   0 0 1 1 1 0 0 1 1 1 1 1

Parent Selection: Biased Roulette Wheel

- For each hypothesis we want to test, spin the roulette wheel to determine the guess



Parent Selection: Additional Selection Criteria

- Elitism
  - Some of the best chromosomes from previous generation replace some of the worst chromosomes from current generation.

- Diversity Measure
  - Fitness ignores diversity.
  - As a result, populations tend to become uniform.
  - Rank-space method
  - Sort population by sum of fitness rank and diversity rank
  - Diversity rank is the result of sorting by the function $1/d^2$

How to use Local Search techniques for path problems

Representation:
Sequence of directions: N, S, E, W

Solution: Gene that leads from S to G

At each decision point in maze, look up next direction.

Path to goal: W \

Fitness Function
+ # steps

Note: make gen

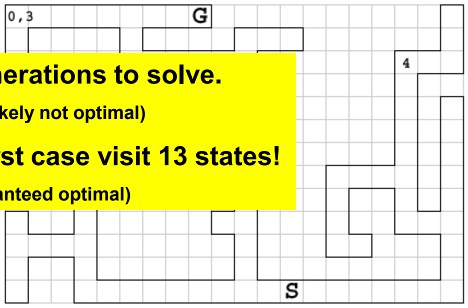
0) EWWNNEWW
f=31 (1,1) 15 moves
1) WSWNEESSESWNENE
f=37
2) SNEESEWWNNNSEWN
f=37
3) EEWWSNNSENWNENW
f=31
4) WEEWSENWSWWSNEN
f=37

GA takes 40 generations to solve.

(and is most likely not optimal)

Heuristic search worst case visit 13 states!

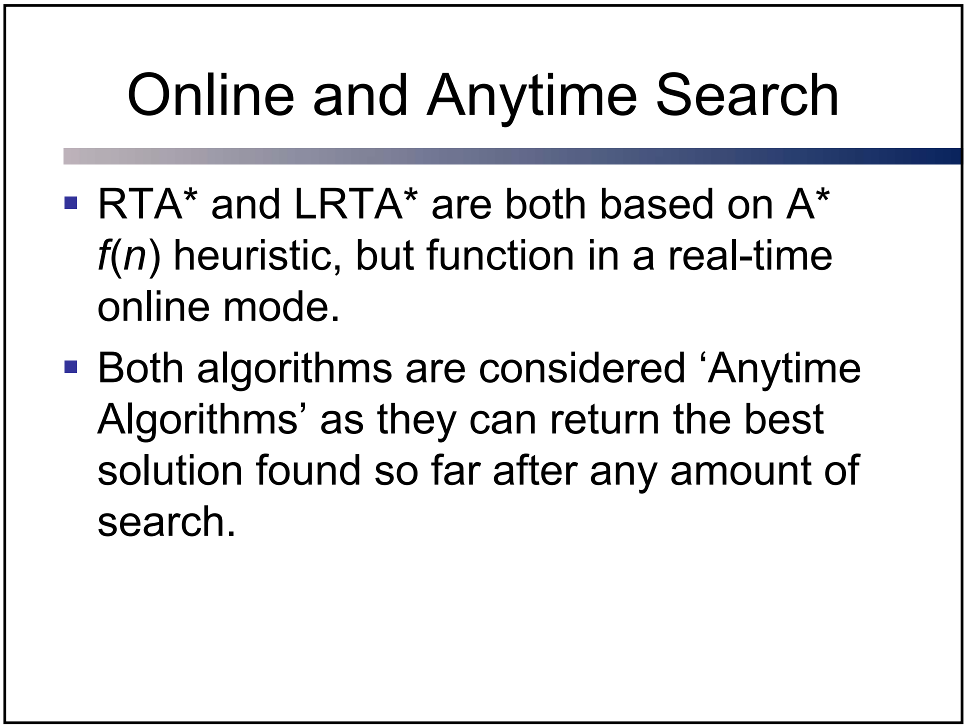
(and is guaranteed optimal)



GA Applications

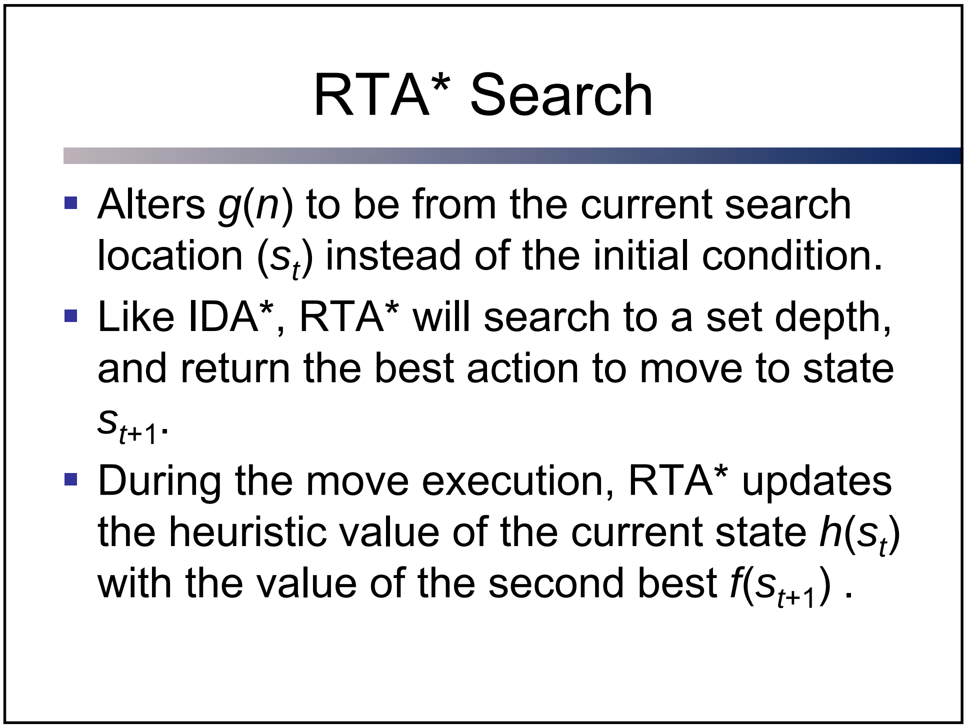
- Control
- Optimization
- Graphic Animation
- Scheduling
- Classifier Systems (load balancing)

- Illustration:
  - https://www.youtube.com/watch?v=uwz8JzrEwWY



Online and Anytime Search

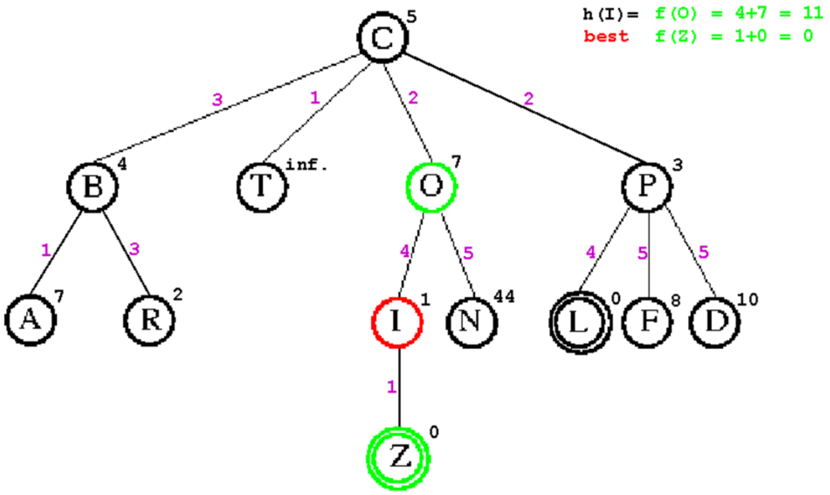
- RTA* and LRTA* are both based on A* *f*(*n*) heuristic, but function in a real-time online mode.
- Both algorithms are considered ‘Anytime Algorithms’ as they can return the best solution found so far after any amount of search.



RTA* Search

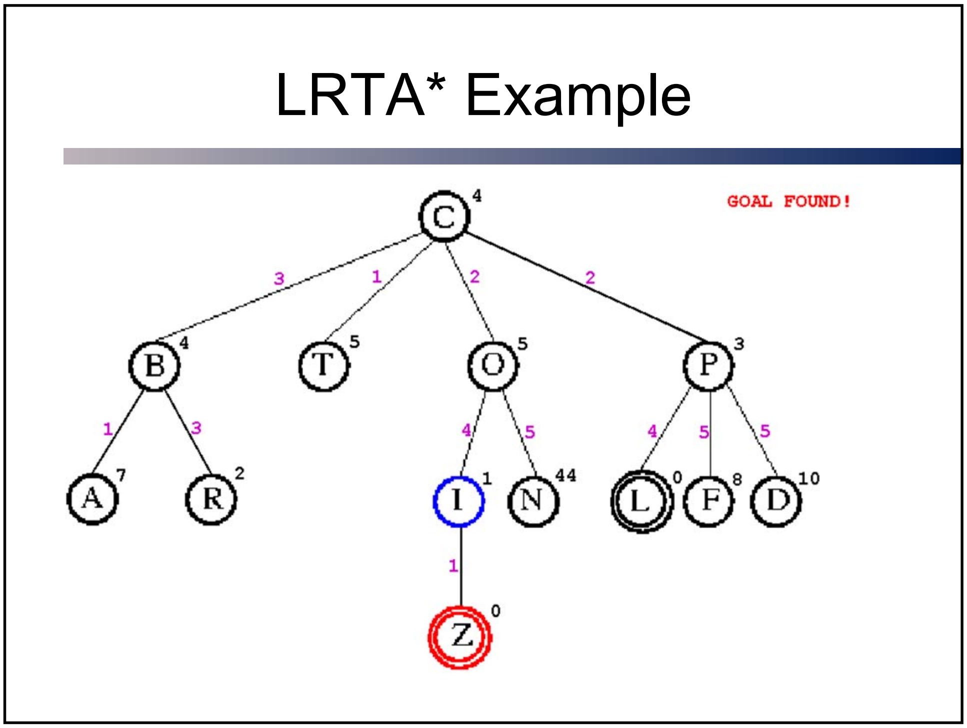
- Alters *g*(*n*) to be from the current search location (s_t) instead of the initial condition.
- Like IDA*, RTA* will search to a set depth, and return the best action to move to state s_{t+1}.
- During the move execution, RTA* updates the heuristic value of the current state $h(s_t)$ with the value of the second best $f(s_{t+1})$.

RTA* Example



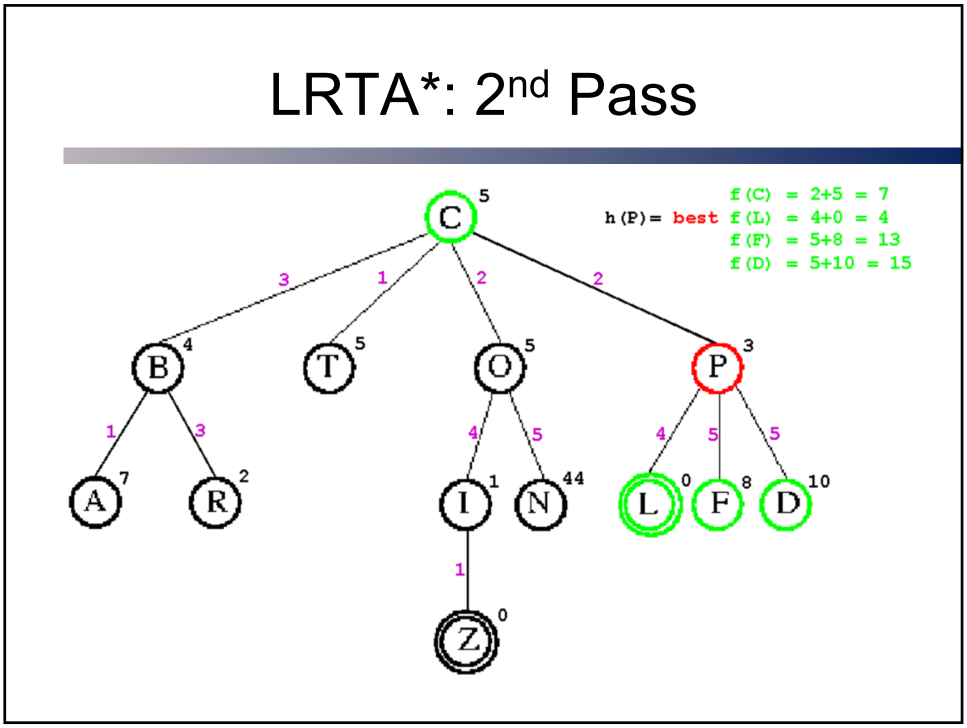
LRTA* Search

- Can the new *h*(*n*) values from RTA* be used for a later search?
  - No, the new *h*(*n*) values now overestimate the distance to the goal.
  - However, if we store the best *f*(*n*) as *h*(*n*) instead of the 2nd best, we can learn the optimal heuristic values.



LRTA* Example

GOAL FOUND!



LRTA*: 2nd Pass

```
          f(C) = 2+5 = 7
h(P)= best f(L) = 4+0 = 4
          f(F) = 5+8 = 13
          f(D) = 5+10 = 15
```

Stochastic Anytime Algorithms

- Stochastic anytime algorithms offer approximate solutions, returning the best solution found so far after a set amount of search
- Based on Monte Carlo techniques
  - Action selection
    - Uniform sampling
    - Heuristic/progressive bias
    - Selective bias
    - Combinations

Upper Confidence Bounds (UCB)

- Given N machines which should we play?
- UCB1
  - Play action j where j is:

$$\max \bar{x}_j + \sqrt{\frac{2\ln n}{n_j}}$$

  - $\bar{x}_j$ average reward of j
  - n_j number of plays of j
  - n total plays

Rollout Based Monte Carlo Search

- MC Search is DFS based and represented recursively

```
(real,action) search(state,depth)
   if (goal state)           ; Test to see if current state
                             ; satisfies the goal
      then SUCCEED           ; Search is complete
   if (leaf)
      then return Evaluation(state)
   action = selectAction(state)
   nextstate = selectChild(state,action)
   q = reward + γ search(nextstate,depth+1)
   Updatevalue(state, action, q depth)
   return q, action
```

- selectAction uses UCB1 strategy
- In non-MDP(Markov decision process) problems, reward is not used
- γ is a decay function 0 < γ < 1
- Evaluation can be based on state or the search tree (Realization Probability Search (RPS))

UCB Applied to Trees (UCT)

- UCB-based rollout-based Monte Carlo search uses the same action selection values at each state in the search tree
  - Video explaining UCB concept: https://www.youtube.com/watch?v=W1p76sq15a8
- UCT maintains a state and action value for each state visited in the tree
- Alternative action selection $(C_p > 0)$

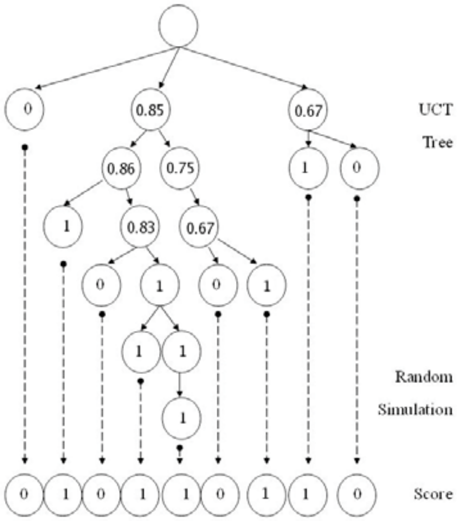
Favors actions that looked good historically

$$\max \bar{x}_j + 2C_p\sqrt{\frac{\ln n}{n_j}}$$

Actions get an exploration bonus that grows w/ *ln(n)*

Source: https://www.youtube.com/watch?v=1HV6uENCs9o

UCT Example

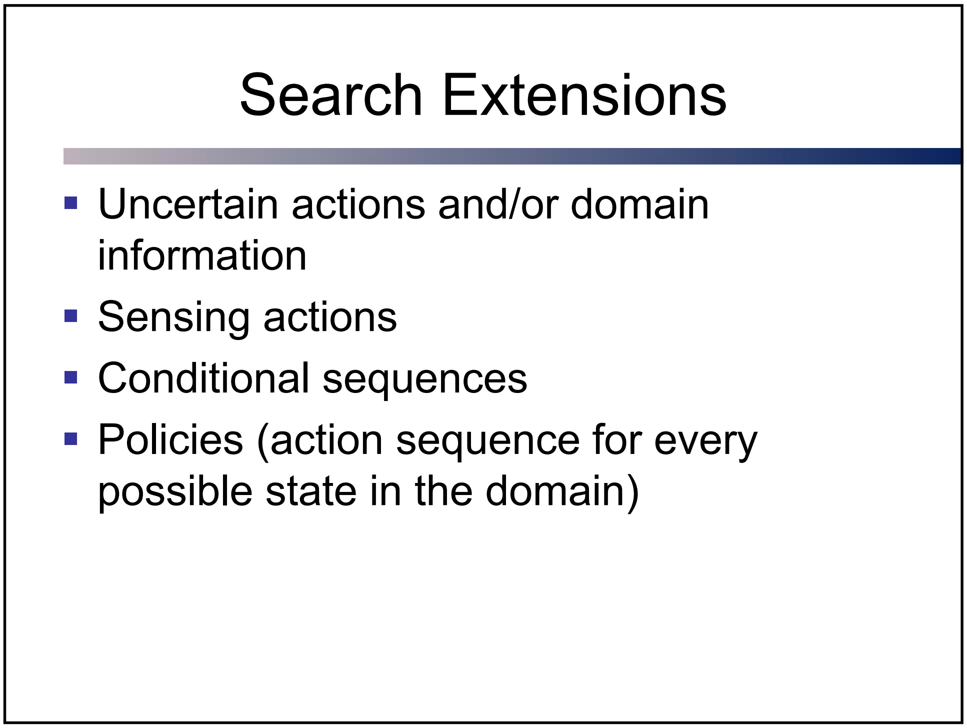


UCT Enhancements

- There is no independence between arms vertically and horizontally, this improves performance when n is small
  - Average results from neighbors (add the term)

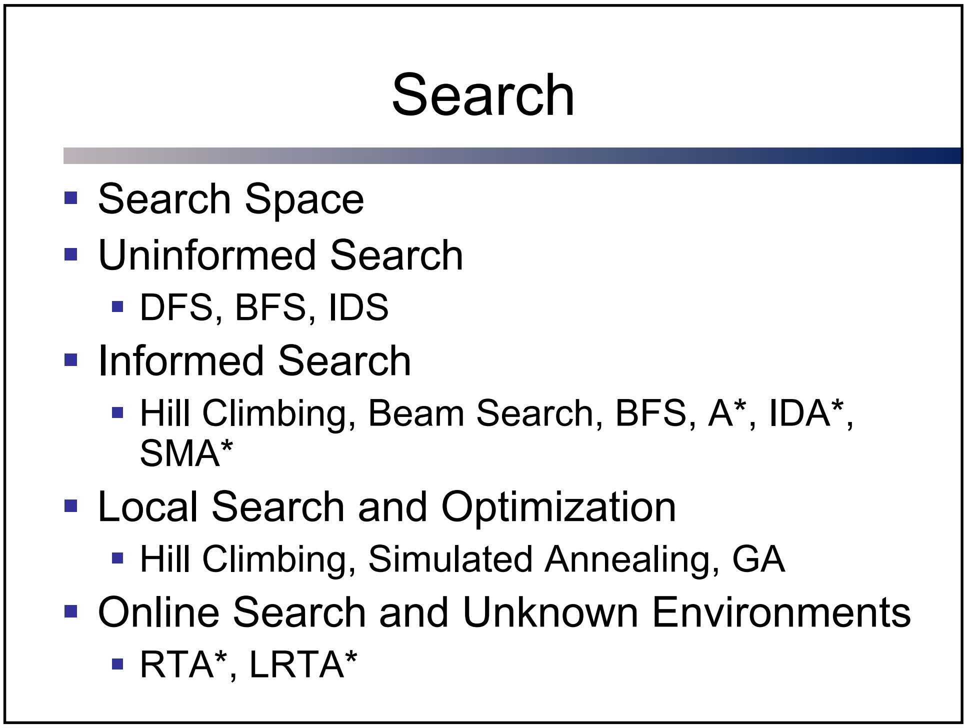
$$\frac{1}{|N_i|}\sum_{j \in N} \bar{x}_j$$

  - Use ancestor results; set a c_i to replace C_p for each move according to $\overline{x_i}$ of its parent



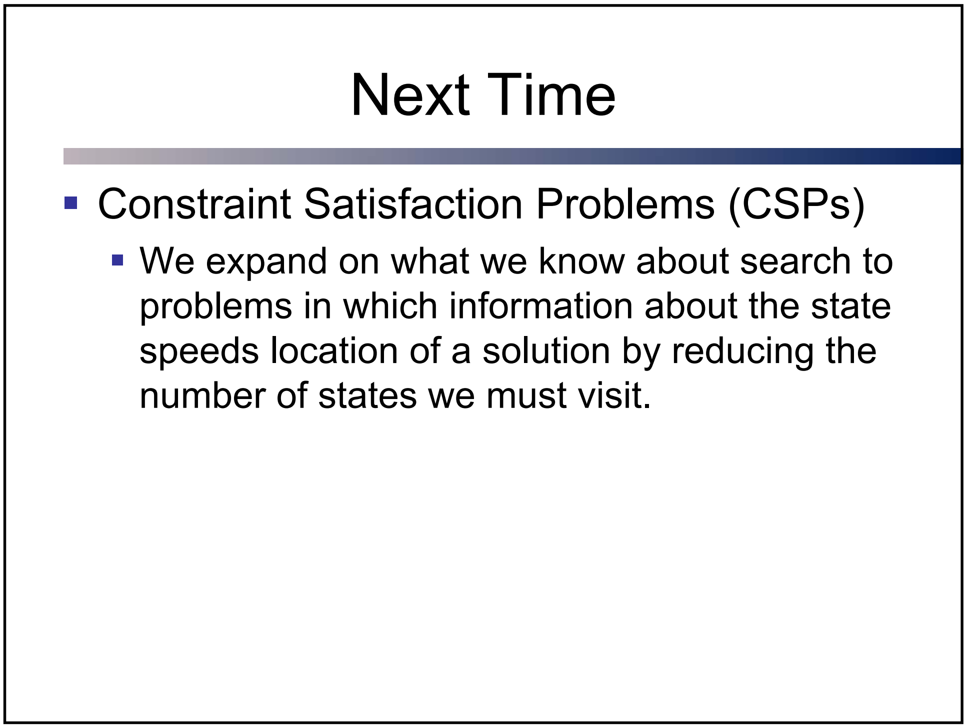
Search Extensions

- Uncertain actions and/or domain information
- Sensing actions
- Conditional sequences
- Policies (action sequence for every possible state in the domain)



Search

- Search Space
- Uninformed Search
  - DFS, BFS, IDS
- Informed Search
  - Hill Climbing, Beam Search, BFS, A*, IDA*, SMA*
- Local Search and Optimization
  - Hill Climbing, Simulated Annealing, GA
- Online Search and Unknown Environments
  - RTA*, LRTA*



Next Time

- Constraint Satisfaction Problems (CSPs)
  - We expand on what we know about search to problems in which information about the state speeds location of a solution by reducing the number of states we must visit.