Classification

Vikram Pudi
vikram@iiit.ac.in
IIIT Hyderabad

Talk Outline

- Introduction
  - Classification Problem
  - Applications
  - Metrics
  - Combining classifiers
- Classification Techniques

The Classification Problem

Outlook	Temp (°F)	Humidity (%)	Windy?	Class
sunny	75	70	true	*play*
sunny	80	90	true	*don't play*
sunny	85	85	false	*don't play*
sunny	72	95	false	*don't play*
sunny	69	70	false	*play*
overcast	72	90	true	*play*
overcast	83	78	false	*play*
overcast	64	65	true	*play*
overcast	81	75	false	*play*
rain	71	80	true	*don't play*
rain	65	70	true	*don't play*
rain	75	80	false	*play*
rain	68	80	false	*play*
rain	70	96	false	*play*
sunny	77	69	true	?
rain	73	76	false	?

Play Outside?

Model relationship between class labels and attributes

e.g. outlook = overcast ⇒ class = *play*

⇒ Assign class labels to new data with *unknown* labels

Applications

- Text classification
  - Classify emails into spam / non-spam
  - Classify web-pages into yahoo-type hierarchy
  - NLP Problems
    - Tagging: Classify words into verbs, nouns, etc.
- Risk management, Fraud detection, Computer intrusion detection
  - Given the properties of a transaction (items purchased, amount, location, customer profile, etc.)
  - Determine if it is a fraud
- Machine learning / pattern recognition applications
  - Vision
  - Speech recognition
  - etc.
- All of science & knowledge is about predicting future in terms of past
  - So classification is a very fundamental problem with ultra-wide scope of applications

Metrics

1. accuracy
2. classification time per new record
3. training time
4. main memory usage (during classification)
5. model size

Accuracy Measure

- Prediction is just like tossing a coin (random variable X)
  - "Head" is "success" in classification; X = 1
  - "tail" is "error"; X = 0
  - X is actually a mapping: {"success": 1, "error" : 0}
- In statistics, a succession of independent events like this is called a *bernoulli process*
  - Accuracy = P(X = 1) = p
  - mean value = μ = E[X] = $p \times 1 + (1-p) \times 0 = p$
  - variance = σ^2 = E[$(X-\mu)^2$] = $p\,(1-p)$
- Confidence intervals: Instead of saying accuracy = 85%, we want to say: accuracy $\in$ [83, 87] with a confidence of 95%

Binomial Distribution

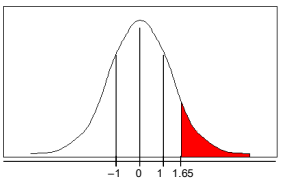
- Treat each classified record as a bernoulli trial
- If there are *n* records, there are *n* independent and identically distributed (iid) bernoulli trials, X_i, i = 1,…,*n*
- Then, the random variable $X = \sum_{i=1,\ldots,n} X_i$ is said to follow a *binomial distribution*
  - $P(X = k) = {}^nC_k\, p^k\, (1-p)^{n-k}$
- Problem: Difficult to compute for large *n*

Normal Distribution

- Continuous distribution with parameters μ (mean), σ^2(variance)
- Probability density:
  $f(x) = (1/\sqrt{(2\pi\sigma^2)})\, \exp(-(x-\mu)^2 / (2\sigma^2))$
- Central limit theorem:
  - Under certain conditions, the distribution of the sum of a *large number* of iid random variables is approximately normal
  - A *binomial distribution* with parameters *n* and *p* is approximately normal for large *n* and *p* not too close to 1 or 0
  - The approximating normal distribution has mean μ = *np* and standard deviation $\sigma^2 = (n\, p\, (1 - p))$

Confidence Intervals

Normal distribution with mean = 0 and variance = 1



Pr[$X \geq z$]	z
0.1%	3.09
0.5%	2.58
1%	2.33
5%	1.65
10%	1.28
20%	0.84
40%	0.25

- E.g. $P[-1.65 \leq X \leq 1.65] = 1 - 2 \times P[X \geq 1.65] = 90\%$
- To use this we have to transform our random variable to have mean = 0 and variance = 1
- Subtract mean from *X* and divide by standard deviation

Estimating Accuracy

- Holdout method
  - Randomly partition data: training set + test set
  - accuracy = |correctly classified points| / |test data points|
- Stratification
  - Ensure each class has approximately equal proportions in both partitions
- Random subsampling
  - Repeat holdout *k* times. Output average accuracy.
- *k*-fold cross-validation
  - Randomly partition data: $S_1, S_2, \ldots, S_k$
  - First, keep S_1 as test set, remaining as training set
  - Next, keep S_2 as test set, remaining as training set, etc.
  - accuracy = |*total* correctly classified points| / |total data points|
- Recommendation:
  - Stratified 10-fold cross-validation. If possible, repeat 10 times and average results. (reduces variance)

Is Accuracy Enough?

- If only 1% population has cancer, then a test for cancer that classifies *all* people as *non-cancer* will have 99% accuracy.
- Instead output a confusion matrix:

Actual/ Estimate	Class 1	Class 2	Class 3
Class 1	90%	5%	5%
Class 2	2%	91%	7%
Class 3	8%	3%	89%

Combining Classifiers

- Get *k* random samples with replacement as training sets (like in random subsampling).

⇒ We get *k* classifiers

- Bagging: Take a *majority vote* for the best class for each new record
- Boosting: Each classifier's vote has a *weight* proportional to its accuracy on training data

⇒ Like a patient taking multiple opinions from several doctors

Talk Outline

- Introduction
- Classification Techniques
  1. Nearest Neighbour Methods
  2. Decision Trees
     - ID3, CART, C4.5, C5.0, SLIQ, SPRINT
  3. Bayesian Methods
     - Naïve Bayes, Bayesian Belief Networks
     - Maximum Entropy Based Approaches
  4. Association Rule Based Approaches
  5. Soft-computing Methods:
     - Genetic Algorithms, Rough Sets, Fuzzy Sets, Neural Networks
  6. Support Vector Machines
  7. Convolutional Neural Networks, Deep Learning

Nearest Neighbour Methods

k-NN, Reverse Nearest Neighbours

k-Nearest Neighbours

- Model = Training data
- Classify record R using the *k* nearest neighbours of R in the training data.
- Most frequent class among *k* NNs
- Distance function could be euclidean
- Use an index structure (e.g. R* tree) to find the *k* NNs efficiently

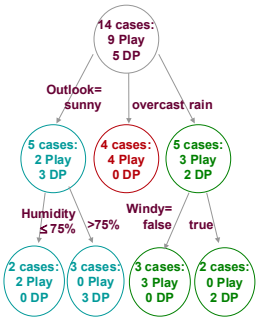
Reverse Nearest Neighbours

- Records which consider *R* as a *k*-NN
- Output most frequent class among RNNs.
- More resilient to outliers.

Decision Trees

Decision Trees

Outlook	Temp (°F)	Humidity (%)	Windy?	Class
sunny	75	70	true	*play*
sunny	80	90	true	*don't play*
sunny	85	85	false	*don't play*
sunny	72	95	false	*don't play*
sunny	69	70	false	*play*
overcast	72	90	true	*play*
overcast	83	78	false	*play*
overcast	64	65	true	*play*
overcast	81	75	false	*play*
rain	71	80	true	*don't play*
rain	65	70	true	*don't play*
rain	75	80	false	*play*
rain	68	80	false	*play*
rain	70	96	false	*play*



Basic Tree Building Algorithm

MakeTree (Training Data D):
    Partition(D)

Partition (Data D):
    if all points in D are in same class: return
    Evaluate splits for each attribute A
    Use best split found to partition D into $D_1, D_2, \ldots, D_n$
    for each D_i:
        Partition (D_i)

ID3, CART

ID3
- Use *information gain* to determine best split
- *gain* = $H(D) - \sum_{i=1\ldots n} P(D_i)\, H(D_i)$
- $H(p_1, p_2, \ldots, p_m) = -\sum_{i=1\ldots m} p_i \log p_i$
- like 20-question game
  - Which attribute is better to look for first: "Is it a living thing?" or "Is it a duster?"

CART
- Only create two children for each node
- Goodness of a split (Φ)
  $\Phi = 2\, P(D_1)\, P(D_2) \sum_{i=1\ldots m} | P(C_j / D_1) - P(C_j / D_2) |$

Shannon's Entropy

- An expt has several possible outcomes
- In N expts, suppose each outcome occurs M times
- This means there are N/M possible outcomes
- To represent each outcome, we need log N/M bits.
  - This generalizes even when all outcomes are not equally frequent.
  - Reason: For an outcome j that occurs M times, there are N/M equi-probable events among which only one cp to j
- Since p_i = M / N, information content of an outcome is -log p_i
- So, expected info content: $H = -\Sigma\, p_i \log p_i$

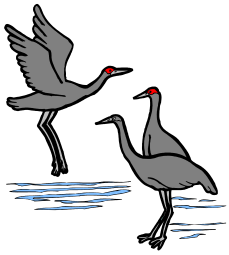
Bayesian Methods

Naïve Bayes

- New data point to classify: $X=(x_1, x_2, \ldots x_m)$
- Strategy:
  - Calculate $P(C_i/X)$ for each class C_i.
  - Select C_i for which $P(C_i/X)$ is maximum

$$P(C_i/X) = P(X/C_i)\, P(C_i) / P(X)$$
$$\propto P(X/C_i)\, P(C_i)$$
$$\propto P(x_1/C_i)\, P(x_2/C_i) \ldots P(x_m/C_i)\, P(C_i)$$

- Naïvely assumes that each x_i is independent
- We represent $P(X/C_i)$ by $P(X)$, etc. when unambiguous



Clustering

Birds of a feather flock together.

Vikram Pudi
vikram@iiit.ac.in
IIIT Hyderabad

The Clustering Problem

Outlook	Temp (°F)	Humidity (%)	Windy?
sunny	75	70	true
sunny	80	90	true
sunny	85	85	false
sunny	72	95	false
sunny	69	70	false
overcast	72	90	true
overcast	73	88	true
overcast	64	65	true
overcast	81	75	false
rain	71	80	true
rain	65	70	true
rain	75	80	false
rain	68	80	false
rain	70	96	false

Find groups of similar records.

Need a function to compute similarity, given 2 input records

⇒ Unsupervised learning

Applications

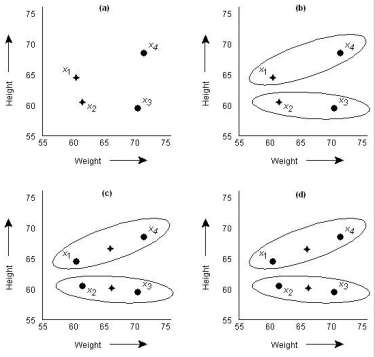
- Targetting similar people or objects
  - Student tutorial groups
  - Hobby groups
  - Health support groups
  - Customer groups for marketing
  - Organizing e-mail
- Spatial clustering
  - Exam centres
  - Locations for a business chain
  - Planning a political strategy

Measurement of similarity

- Nominal (categorical) variables
  d(x,y) = 1 - m/n
  m = no of matches among n attributes, or
  m = sum of weights of matching attributes, and n is the sum of weights of all attributes
- Numeric variables
  - Euclidean, manhattan, minkowski,…
  - Ordinal
    - z = (rank-1)/(M-1) where M is maximum rank
- Above are examples
  - Similarity is ultimately application dependent
  - Requires various kinds of preprocessing
    - Scaling: Convert all attributes to have same range
    - z-score: z = (value-mean)/m
      where m is the mean absolute deviation

Partitioning technique: *k*-Means

- Initial *k* means = random records
- Iterate as long as clusters change:
  - Put each record X in the cluster to whose mean it is closest
  - Recompute means as the average of all points in each cluster



Evaluating Clustering Quality

- Minimize squared error
  Here m_i is the mean (or other centre) of cluster i
- Can also use absolute error
- Can be used to find best initial random means in *k*-means.

$$E = \sum_{i=1}^{N} \sum_{x \in C_i} d(x, m_i)^2$$

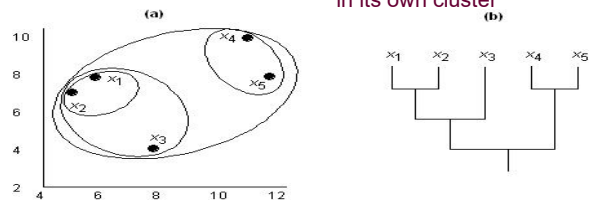
Hierarchical Methods

Agglomerative (e.g. AGNES):

- Start: Each point in separate cluster
- Merge 2 closest clusters
- Repeat until all records are in 1 cluster.

Divisive (e.g. DIANA)

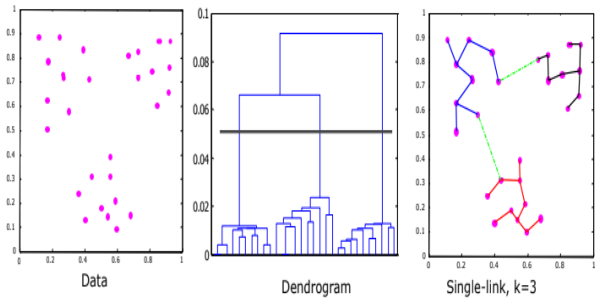
- Start: All points in 1 cluster
- Find most extreme points in each cluster.
- Regroup points based on closest extreme point
- Repeat until each record is in its own cluster



Measuring Cluster Distances

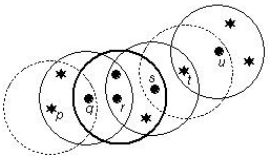
- Single link: Minimum distance
- Complete link: Maximum distance
- Average link: Average distance

Single Link Algorithm



Density-based Methods: e.g. DBSCAN

- Neighbourhood: Records within distance of ε from given record.
- Core point: Record whose neighbourhood contains at least μ records.
- Find all core points and create a cluster for each of them.
- If core point Y is in the neighbourhood of core point X, then merge the clusters of X and Y.
- Repeat above step for all core points until clusters do not change.



Mining Outliers using Clustering

- Outliers are data points that deviate significantly from the norm.
- Useful in fraud detection, error detection (in data cleaning), etc.
- Technique:
  - Apply any clustering algorithm
  - Treat clusters of very small size as containing only outliers

Solving problems by searching

From AIMA slides

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

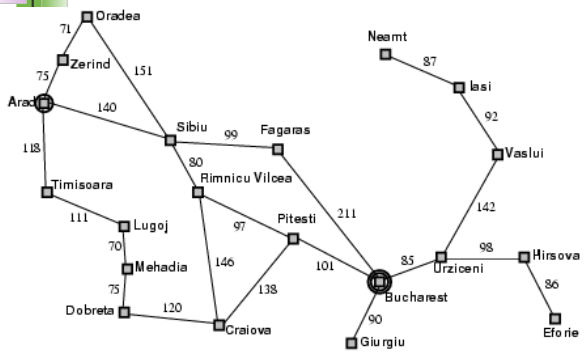
Problem-solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```


Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania

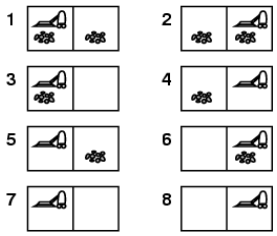


Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave} search, execution
- Unknown state space → exploration problem

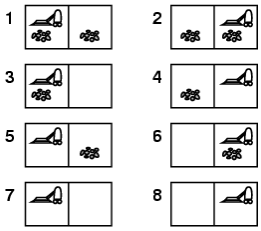
Example: vacuum world

- Single-state, start in #5.
  Solution?



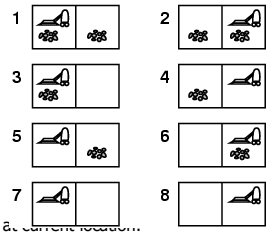
Example: vacuum world

- Single-state, start in #5.
  Solution? *[Right, Suck]*
- Sensorless, start in *{1,2,3,4,5,6,7,8}* e.g., *Right* goes to *{2,4,6,8}*
  Solution?



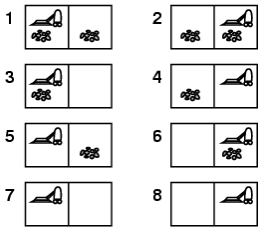
Example: vacuum world

- Sensorless, start in *{1,2,3,4,5,6,7,8}* e.g., *Right* goes to *{2,4,6,8}*
  Solution?
  [Right,Suck,Left,Suck]
- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean]*, i.e., start in #5 or #7
    Solution?



Example: vacuum world

- Sensorless, start in *{1,2,3,4,5,6,7,8}* e.g., *Right* goes to *{2,4,6,8}*
  Solution?
  [Right,Suck,Left,Suck]
- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean]*, i.e., start in #5 or #7
    Solution? *[Right, **if** dirt **then** Suck]*



Single-state problem formulation

A problem is defined by four items:

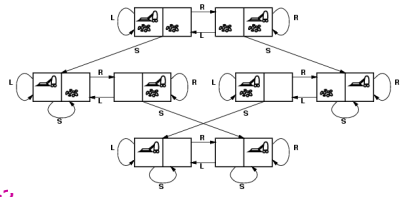
1. initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad)* = *{<Arad → Zerind, Zerind>, … }*
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

- A solution is a sequence of actions leading from the initial state to a goal state

Selecting a state space

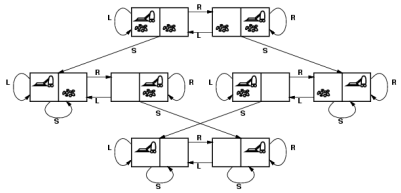
- Real world is absurdly complex
  → state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

Vacuum world state space graph



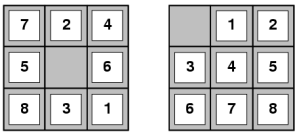
- states?
- actions?
- goal test?
- path cost?

Vacuum world state space graph



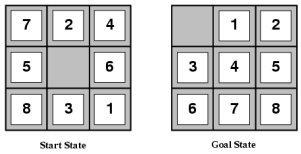
- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



- states?
- actions?
- goal test?
- path cost?

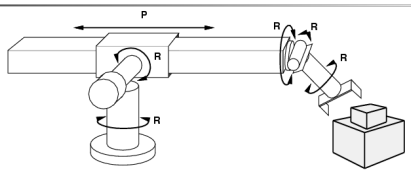
Example: The 8-puzzle



- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

[Note: optimal solution of *n*-Puzzle family is NP-hard]

Example: robotic assembly



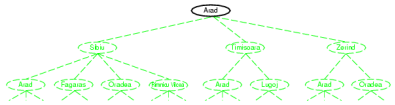
- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

Tree search algorithms

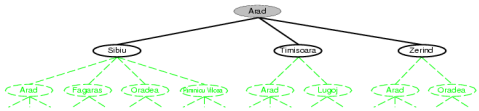
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

```
function TREE-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
```

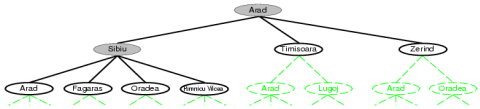
Tree search example



Tree search example



Tree search example

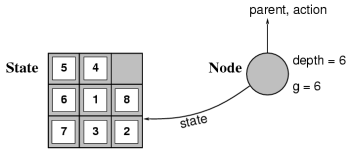


Implementation: general tree search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERT ALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```


Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth



- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

Search strategies

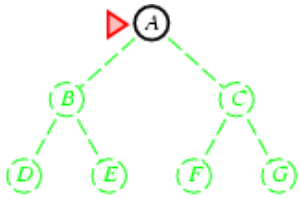
- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

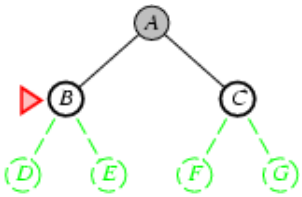
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



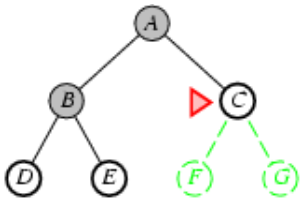
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



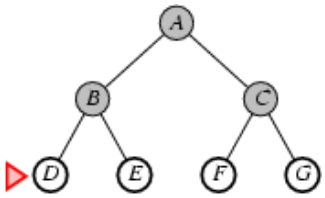
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)

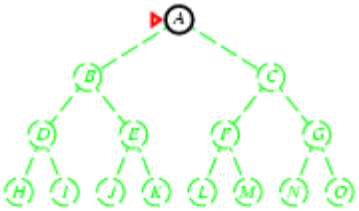
- Space is the bigger problem (more than time)

Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\text{ceiling}(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\text{ceiling}(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

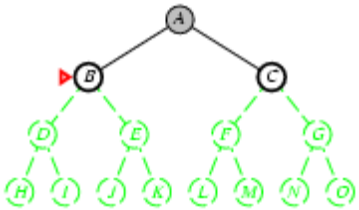
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



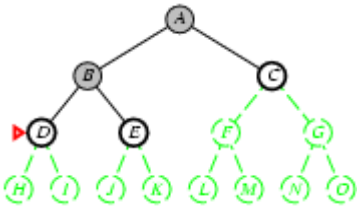
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front
  -
-



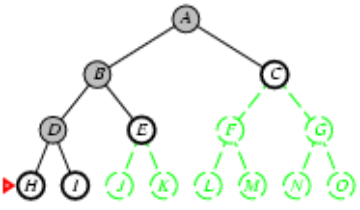
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



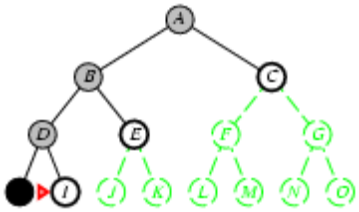
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



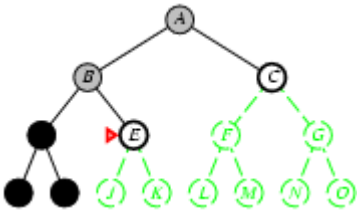
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



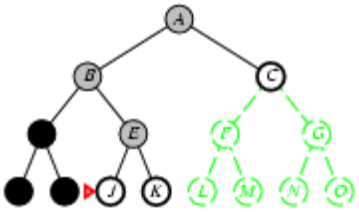
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



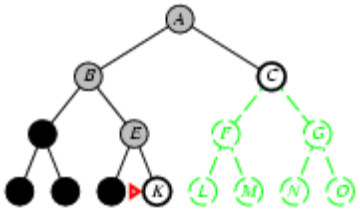
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



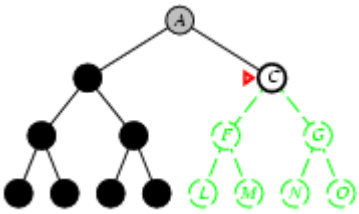
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



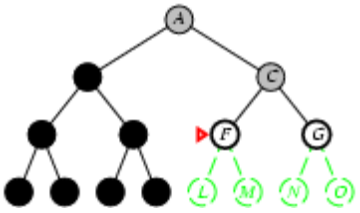
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



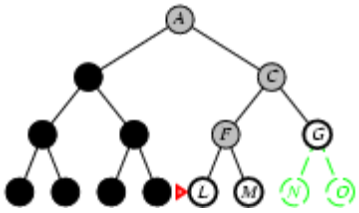
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



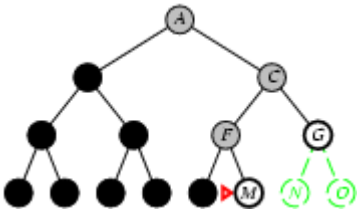
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    → complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No

Depth-limited search

= depth-first search with depth limit *l*
i.e., nodes at depth *l* have no successors

- Recursive implementation:

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

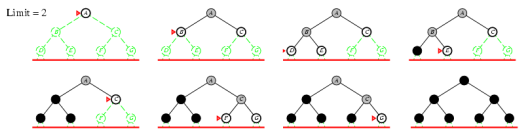

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
  inputs: problem, a problem
  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH( problem, depth)
    if result ≠ cutoff then return result
```

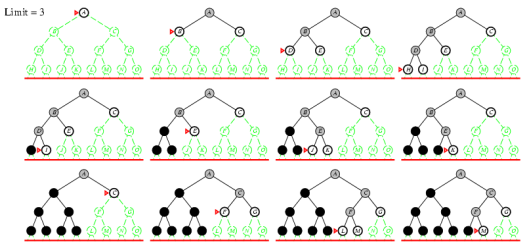

Iterative deepening search *l* =0

Iterative deepening search *l* =1

Iterative deepening search l =2



Iterative deepening search l =3



Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

  $N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

  $N_{IDS} = (d+1)b^0 + d\,b\text{^}1 + (d-1)b\text{^}2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$

- For *b = 10, d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
- Overhead = (123,456 - 111,111)/111,111 = 11%

Properties of iterative deepening search

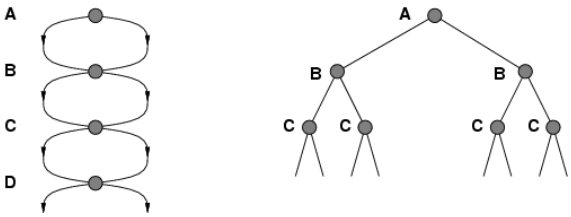
- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!



Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```


Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

Informed search algorithms

From AIMA Slides

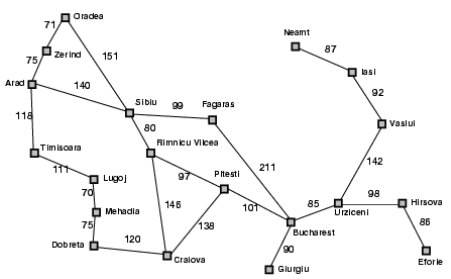
Outline

- Best-first search
- Greedy best-first search
- A* search
- Heuristics
- Local search algorithms
- Hill-climbing search
- Simulated annealing search
- Local beam search
- Genetic algorithms

Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - estimate of "desirability"
  - → Expand most desirable unexpanded node
- Implementation:
  Order the nodes in fringe in decreasing order of desirability
- Special cases:
  - greedy best-first search
  - A* search

Romania with step costs in km



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search

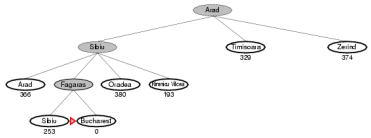
- Evaluation function *f(n)* = *h(n)* (heuristic)
- = estimate of cost from *n* to *goal*
- e.g., $h_{SLD}(n)$ = straight-line distance from *n* to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal

Greedy best-first search example

Greedy best-first search example

Greedy best-first search example

Greedy best-first search example



Properties of greedy best-first search

- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

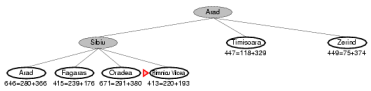
A* search

- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal

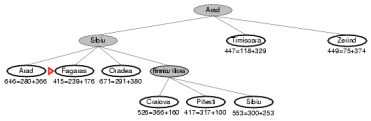
A* search example

A* search example

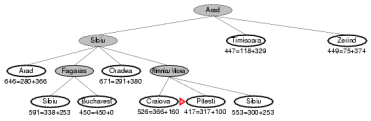
A* search example



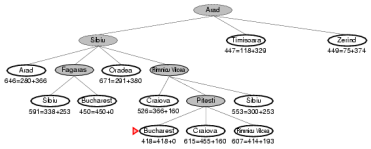
A* search example



A* search example



A* search example

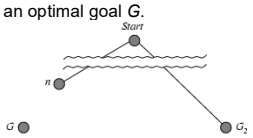


Admissible heuristics

- A heuristic *h(n)* is admissible if for every node *n*, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from *n*.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If *h(n)* is admissible, A* using `TREE-SEARCH` is optimal

Optimality of A* (proof)

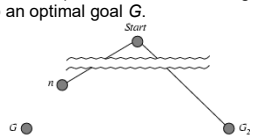
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above

Optimality of A* (proof)

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



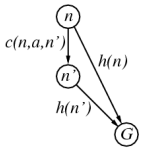
- $f(G_2)$ > $f(G)$ from above
- $h(n)$ ≤ $h^*(n)$ since h is admissible
- $g(n) + h(n)$ ≤ $g(n) + h^*(n)$
- $f(n)$ ≤ $f(G)$

Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion

Consistent heuristics

- A heuristic is consistent if for every node n, every successor n' of n generated by any action a,

$h(n) \le c(n,a,n') + h(n')$



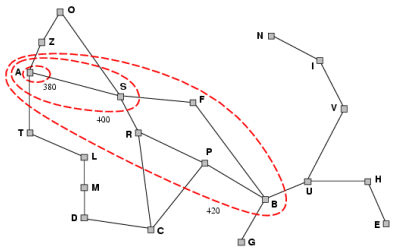
- If h is consistent, we have

$f(n') = g(n') + h(n')$
$= g(n) + c(n,a,n') + h(n')$
$\ge g(n) + h(n)$
$= f(n)$

- i.e., $f(n)$ is non-decreasing along any path.
- Theorem: If $h(n)$ is consistent, A* using GRAPH-SEARCH is optimal

Optimality of A*

- A* expands nodes in order of increasing f value
- Gradually adds "f-contours" of nodes
- Contour i has all nodes with $f=f_i$, where $f_i < f_{i+1}$



Properties of A*

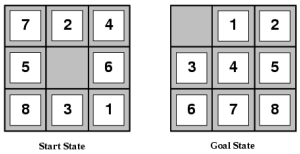
- Complete? Yes (unless there are infinitely many nodes with f ≤ $f(G)$)
- Time? Exponential
- Space? Keeps all nodes in memory
- Optimal? Yes

Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



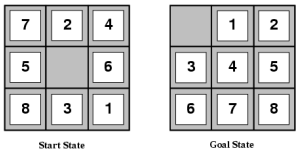
- $h_1(S)$ = ?
- $h_2(S)$ = ?

Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



- $h_1(S)$ = ? 8
- $h_2(S)$ = ? 3+1+2+2+2+3+3+2 = 18

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)
- then h_2 dominates h_1
- h_2 is better for search

- Typical search costs (average number of nodes expanded):

- *d=12* IDS = 3,644,035 nodes
  A*(h_1) = 227 nodes
  A*(h_2) = 73 nodes
- *d=24* IDS = too many nodes
  A*(h_1) = 39,135 nodes
  A*(h_2) = 1,641 nodes

Relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution

- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens

- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

Example: *n*-queens

- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

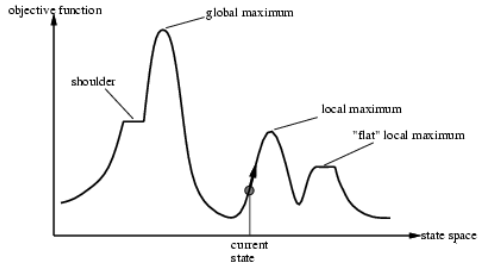
Hill-climbing search

- "Absent-minded blind man climbs a hill"
- Will he reach the highest peak?

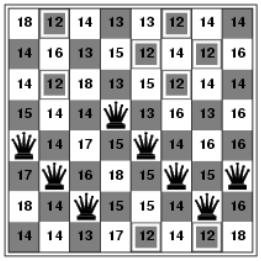
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node
  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
```

Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima

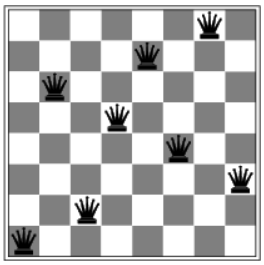


Hill-climbing search: 8-queens problem



- *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- *h* = *17* for the above state

Hill-climbing search: 8-queens problem



- A local minimum with *h* = *1*

Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps
  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```

Properties of simulated annealing search

- One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
- Widely used in VLSI layout, airline scheduling, etc

Local beam search

- Keep track of *k* states rather than just one
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.

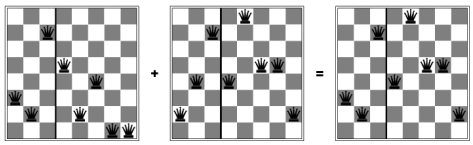
Genetic algorithms

- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

Genetic algorithms

- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

Genetic algorithms



Adversarial Search

From AIMA Slides

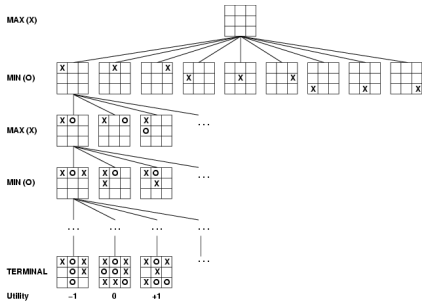
Outline

- Optimal decisions
- α-β pruning
- Imperfect, real-time decisions

Games vs. search problems

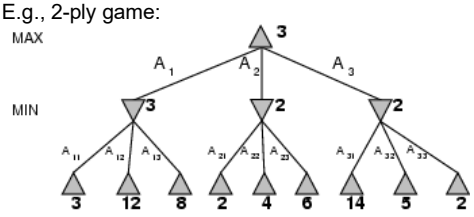
- "Unpredictable" opponent → specifying a move for every possible opponent reply
- Time limits → unlikely to find goal, must approximate

Game tree (2-player, deterministic, turns)



Minimax

- Perfect play for deterministic games
- Idea: choose move to position with highest minimax value
  = best achievable payoff against best play
- E.g., 2-ply game:



Minimax algorithm

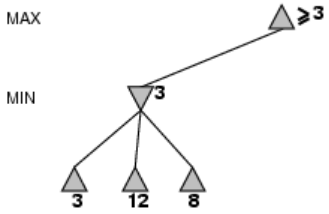
```
function MINIMAX-DECISION(state) returns an action
  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
  return v
```

Properties of minimax

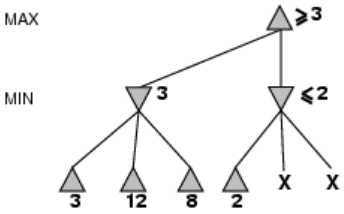
- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)

- For chess, b ≈ 35, m ≈100 for "reasonable" games
  → exact solution completely infeasible

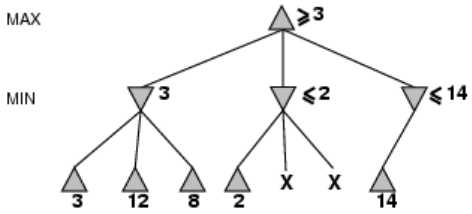
α-β pruning example



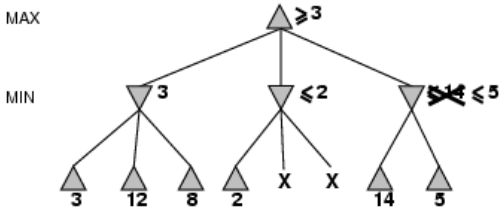
α-β pruning example



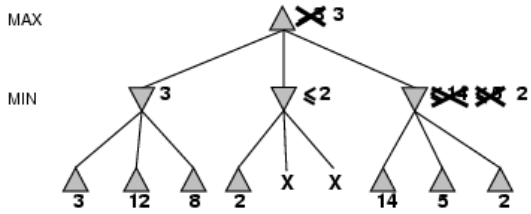
α-β pruning example



α-β pruning example



α-β pruning example

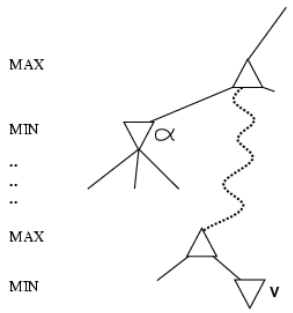


Properties of α-β

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity = $O(b^{m/2})$
  → doubles depth of search
- A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Why is it called α-β?

- α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for *max*
- If *v* is worse than α, *max* will avoid it
  → prune that branch
- Define β similarly for *min*



The α-β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

The α-β algorithm

```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Resource limits

Suppose we have 100 secs, explore 10^4 nodes/sec
→ 10^6 nodes per move

Standard approach:

- cutoff test:
  e.g., depth limit (perhaps add quiescence search)
- evaluation function
  = estimated desirability of position

Evaluation functions

- For chess, typically linear weighted sum of features
  $Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
- e.g., $w_1 = 9$ with
  $f_1(s)$ = (number of white queens) – (number of black queens), etc.

Cutting off search

MinimaxCutoff is identical to *MinimaxValue* except

1. *Terminal?* is replaced by *Cutoff?*
2. *Utility* is replaced by *Eval*

Does it work in practice?

$b^m = 10^6$, b=35 → m=4

4-ply lookahead is a hopeless chess player!

- 4-ply ≈ human novice
- 8-ply ≈ typical PC, human master
- 12-ply ≈ Deep Blue, Kasparov

Deterministic games in practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.
- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.
- Othello: human champions refuse to compete against computers, who are too good.
- Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

Summary

- Games are fun to work on!
- They illustrate several important points about AI
- perfection is unattainable → must approximate
- good idea to think about what to think about