



Artificial Intelligence

Author: Guanzhou (Jose) Hu 胡冠洲 @ MIT 6.034

Teacher: [Kimberle Koile](#) & other 6.034 staff

NOTE - Some uncited figures come from reference notes / quizzes.

Artificial Intelligence

Introduction

- What is AI?

- History of AI

- Problem Solving Strategies

Rule-Based Systems

- Forward Chaining

- Backward Chaining

- Example

Basic Searching

- Deterministic Searching

- Using Heuristics

- Example

Games & Adversarial Searching

- Static Evaluation

- Minimax (Max-Min Steps)

- α, β Pruning

- Progressive Deepening

- Example

Constraints Solving Problems

- Depth-First Solver

- Optimizations with Reductions

- Example

Learning Methods

- Learning by Nearest Neighbors

- Learning by Building Identification Trees

- Example

Basic Neural Nets

- Boolean Logical Neural Nets

- Activation Functions

- Performance Measurement

- Forward Propagation

- Backward Propagation

Support Vector Machines (SVMs)

- Drawing Decision Boundary

- Mathematical Constraints & Calculations

- Transformation & Kernels

Bayes Nets

- Bayesian Network

- Probability & Hypothesis

- Checking Independence

Boosting

- Adaboost

Introduction

What is AI?

AI (Artificial Intelligence) is about artificial ways to achieve intelligence. Operational definition: "Architectures" that employ "Methods" enabled by "Constraints" exposed by "Representations" that support "Models" of perception, thinking, and actions.

Examples: So many, needless to put here ;)

In two aspects, AI is important:

- Engineering: build programs that solve problems
- Science: understand human intelligence, build computational models of human intelligence

Need to clarify: AI is not all about Machine Learning (neither this course)!

History of AI

Wikipedia: [History of Artificial Intelligence](#). (1956 ~ present, and future)

Important Points:

1. Representation right = Almost done
2. Rumplestiltskin principle: once you name something, you have power over it
3. Simple things \neq Trivial things

Problem Solving Strategies

Reducing the problem: Build the *goal tree* (with and/or nodes) \rightarrow Look for a solution \rightarrow *Evaluation* of performance.

Fully exploit the power of:

- Problem decomposition
- Building (mathematical / executable) models
- ...

Rule-Based Systems

A **Rule** is an IF-THEN statement in the form:

```

1  cousin_rule =
2  IF:      (antecedent)
3      AND ( A is parent of B,
4            C is parent of D,
5            A and C are siblings )
6  THEN:    (consequent)
7      B and D are cousins.
8  DELETE:  (side-effect)
9      /

```

A **Knowledge Base** is the set of rules. A **Database** is the collection of all the statements we currently have, which may dynamically get updated as we apply rules.

Forward Chaining

Chain from *Facts* to *Goals*:

1. **Match**: Find Rules that can apply on our current Database (i.e., antecedent accords, no matter whether the consequent exists or not), and bind the variables in it;
2. **Fire**: Pick a matching Rule, check whether the consequent exists in current Database. If not, apply the rule, add the consequent into our Base, and apply the side-effect if there is one;
3. Repeat from 1. until nothing more can be done.

Properties:

- With no NOT / DELETE clauses, a matching Rule must always continue to match
- Without DELETE clauses, a fired Rule must add something into the Database

Backward Chaining

Chain from *Goal* to *Facts*, to see whether a **Hypothesis** is true, and what rules make it true. Essentially is to recursively build an AND-OR tree, where the root is our Hypothesis.

Assumptions:

- At a node, first tries to match in the list of existing assertions; If no matching assertions, then tries to find a rule with matching consequent (this introduces a subtree); If no matching consequent, then raise false
- Always pick assertion / rule / antecedent from the list in order, and append new ones at the bottom
- *Short circuiting*: whenever a branch of an AND is false / a branch of an OR is true, the rest branches are skipped

Example

An example from [2012 Quiz 1](#):

Rules & Initial assertions:

Rules:

```

P0 IF  '(?x) is in 6.034'
    THEN  '(?x) is charismatic'

P1 IF  (OR  '(?x) is nimble',
           '(?x) takes dance lessons')
    THEN  '(?x) can dance like Mick Jagger'

P2 IF  (OR  '(?x) is a rebel',
           (AND  '(?x) has money',
                 '(?x) chooses style over comfort'))
    THEN  '(?x) wears leather'

P3 IF  (AND  '(?x) is related to Lady Gaga',
           (NOT  '(?x) goes to MIT'))
    THEN  '(?x) chooses style over comfort'

P4 IF  (OR  (AND  '(?x) wears leather',
                  '(?x) can dance like Mick Jagger'),
           (AND  '(?x) is musically gifted',
                  '(?x) is charismatic'))
    THEN  '(?x) should become a rockstar'

```

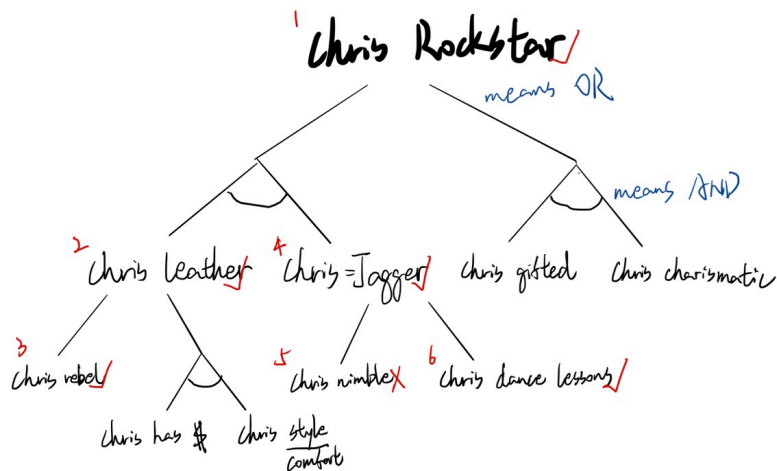
Assertions:

```

A0: (Alison is in 6.034)
A1: (Brad is in 6.034)
A2: (Alison is musically gifted)
A3: (Chris is a rebel)
A4: (Chris takes dance lessons)

```

To **backward chain** to test the hypothesis "Chris should become a rockstar", the constructed goal tree is as follows:



And thus the rules the backward chainer looks is the following in order:

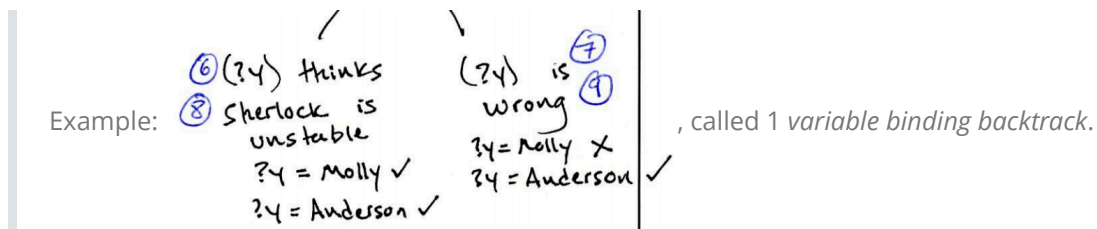
1. Chris should become a rockstar
2. Chris wears leather
3. Chris is a rebel
4. Chris can dance like Mick Jagger
5. Chris is nimble
6. Chris takes dance lessons

Running a full **forward chaining** on the initial assertions, the procedure is:

Iteration	Rules Matched	Rule Fired	New Assertion Added
1	P0, P1, P2	P0	Alison is charismatic
2	P0, P1, P2, P4	P0	Brad is charismatic
3	P0, P1, P2, P4	P1	Chris can dance like Mick Jagger
4	P0, P1, P2, P4	P2	Chris wears leather
5	P0, P1, P2, P4	P4	Alison should become a rockstar
6	P0, P1, P2, P4	P4	Chris should become a rockstar

Additional notes:

- When there is multiple-binding rules `... (?x) ... (?y) ...`, some variables might not get binded at first look-up branch (leave them as unbinded variable, e.g., `(?y)`); after that branch touches the bottom, it may finally be binded to a value so the following sibling antecedent branches should be concrete



- Be careful of NOT / DELETE clauses

Basic Searching

Searching is a class of algorithms that allow exploration of *abstract plans* (= sequences of actions).

- Search tree
- Extended list*: nodes that have been *determined*; when to extend again, avoid adding those paths stepping on these nodes again
- Backtrack*: when deadend is reached, go back one step and iterate one the next extension path choice

Deterministic Searching

- British Museum Search*: from $n = 1 \rightarrow max$, enumerate & check whether there is a solution of scale n
- BFS, DFS* (queue v.s. stack + avoid duplication with *extended list*)
- Branch & Bound (BB)*: in this course an **"extension"** means a new path (instead of a new node to explore), so BB is essentially *Dijkstra*
 - I don't know why they call it BB; for me *pruning* is BB ;(
 - Naturally, BB cannot handle negative edge weights

Using Heuristics

Heuristics! Use them as an estimate of which next step is better to pick up first. A heuristic function h should be:

- **Admissible:** $H_A \leq$ actual true distance from goal; AND
- **Consistent (Monotone):** $|H_A - H_B| \leq$ actual distance between A, B , where they are neighbors;
 - If using an extended list, the heuristic must be consistent
 - o.w., cannot ensure that the solution is optimal

Hill Climbing is different from *Best First*: it stays on the current node head

Beam Search: only top *beam-width* nodes at each level maintained in the agenda

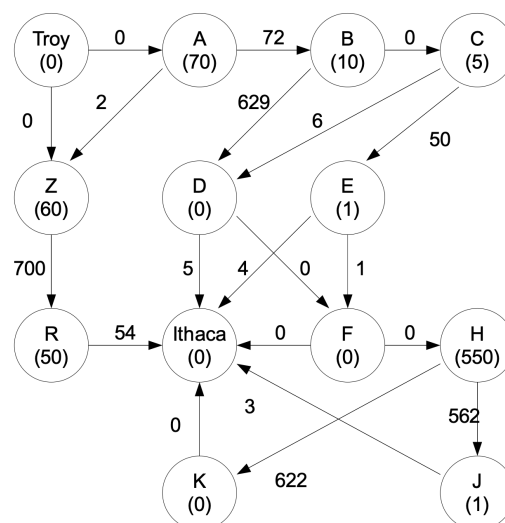
A* Search: omitted

Algorithm	Extend which one in queue?	How to add extensions into queue?	Backtrack?	Extended list?		Possible path guaranteed?	Optimal (edge len considered)?
DFS	1st	front	✓	✓ / ✗		✓	✗
BFS	1st	back	✓	✓ / ✗		✓	✗
Branch & Bound	best by G	anywhere	✓	✓ / ✗		✓	✓
Hill Climbing	1st	sort by H , then front	✓ / ✗	✗		✓ / ✗	✗
Best First	best by H	anywhere	✓	✗		✓	✗
Beam	keep top <i>beam-width</i> ones in each depth by H , then best by H	anywhere	✗	✗		✗	✗
A*	best by $G + H$	anywhere	✓	✓		✓	✓

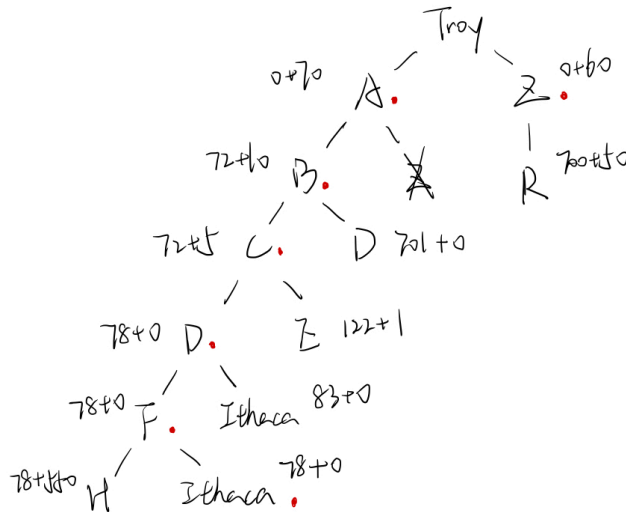
Example

Example from 2010 Quiz 1:

Given the following graph with start node "Troy" → goal "Ithaca":



The **A*** search tree w/ extended list is as follows:



It **expands nodes in the order**: Troy, Z, A, B, C, D, F, Ithaca.

It gives an optimal path: Troy \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow F \rightarrow Ithaca.

Games & Adversarial Searching

In games, using Rule-based Systems results in lack in global & long-term view, while basic searching algorithm only involve one single target. (It's like players A & B both want A to win.) That's why we need Adversarial Searching.

Static Evaluation

The computation of the score of a certain *State (Snapshot)* is called **Static Evaluation**. When player A is evaluating:

- A's advantage \Rightarrow Score higher
- B's advantage \Rightarrow Score lower
- Tie \Rightarrow Score = 0

For endgame states, there's normally a fixed evaluation rule so that, say, A wins = 1000, B wins = -1000, and Tie = 0.

For non-endgame states, the game is not over yet. We will need a heuristic function on the state to give an estimated score.

Minimax (Max-Min Steps)

Build a *Game Tree*, where each node is a state. Root is the current state, and children of a node are the states we can go to in that turn from that node. Leaves are all endgame states.

Layers interleave as MAX / MIN, representing that we forecast MAX / MIN player's turn. From top down, **MAX player** tries to pick the move towards maximum score, when **MIN player** tries to minimize. Essentially it is a DFS with the goal flipping at each depth.

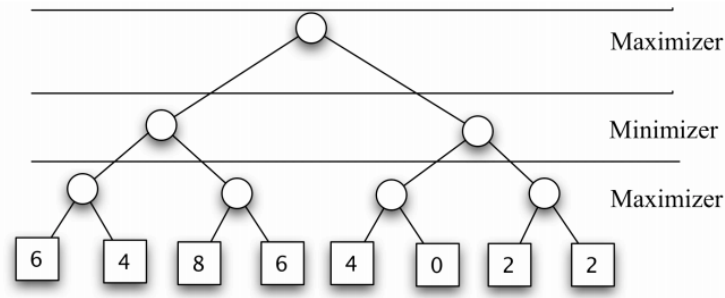


Figure from [Prof. Bob Berwick's note](#).

```

1 def minimax_till_endgame(state, maximize):
2     # Base case reached.
3     if state.is_game_over():
4         return ([state], state.get_endgame_score(maximize), 1)
5     # Recurse on all children.
6     children = [minimax_endgame_search(s, not maximize)
7                 for s in state.generate_next_states()]
8     # If i'm maximizing, pick the maximum of children's score results; o.w., pick
the minimum.
9     if maximize:
10        best_moves, best_score, _ = max(children, key=lambda tup: tup[1])
11    else:
12        best_moves, best_score, _ = min(children, key=lambda tup: tup[1])
13    # Number of evaluations done is the sum up of all branches.
14    num_evals = sum(map(lambda tup: tup[2], children))
15    # Enqueue current state to the moves.
16    return ([state] + best_moves, best_score, num_evals)

```

We will get a `best_moves` list from this algorithm which ideally leads to that `best_score`. Make the first move in list.

The problem with `minimax` till endgame (leaves) is that we have to search until the leaf nodes, which is absolutely not feasible for most of the games. So we should give a depth limit and search only up to that limit. If we do not reach the leaves at depth limit, use heuristics to get scores.

```

1 def minimax_search(state, maximize, heuristic_fn=always_zero, depth_limit=INF):
2     # Base case reached.
3     if state.is_game_over():
4         return ([state], state.get_endgame_score(maximize), 1)
5     elif depth_limit == 0:
6         return ([state], heuristic_fn(state, maximize), 1)
7     # Recurse on all children.
8     children = [minimax_search(s, heuristic_fn, depth_limit - 1, not maximize)
9                 for s in state.generate_next_states()]
10    # ...

```

α, β Pruning

With depth increasing, search time increases exponentially. There is one thing that we can optimize: if the current subtree we are in already conflicts with the results from previous subtrees (i.e., the best score I can get from my children is worse than the best of my previous siblings), then simply **prune** this subtree off. The play will never go into this subtree, since the opponent will not allow him to do so in the last turn.

We maintain two values α, β :

- α is the score that the MAX player will at least get, from the nodes already searched
- β is the score that the MIN player will at most get, ...

Whenever $\alpha \geq \beta$, we no longer need to explore this subtree.

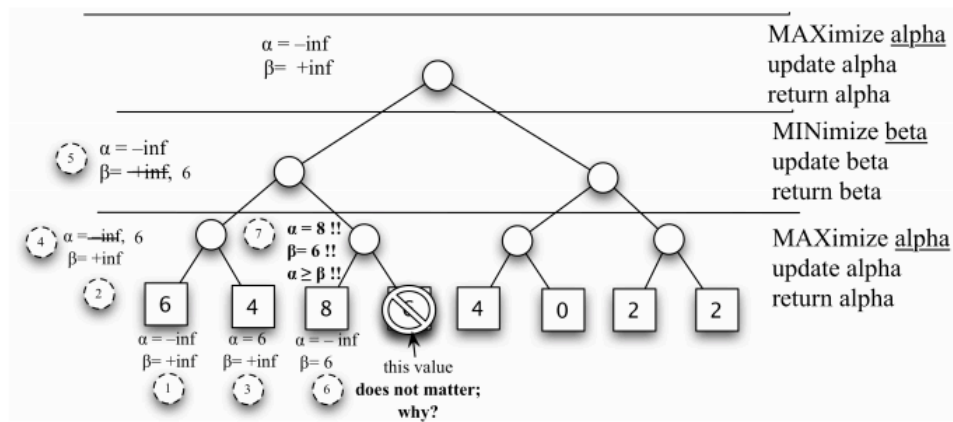


Figure from [Prof. Bob Berwick's note](#).

```

1 def minimax_search_alphabeta(state, alpha=-INF, beta=INF,
2 heuristic_fn=always_zero,
3                               depth_limit=INF, maximize=True):
4     # Base case reached.
5     if state.is_game_over():
6         return ([state], state.get_endgame_score(maximize), 1)
7     elif depth_limit == 0:
8         return ([state], heuristic_fn(state, maximize), 1)
9     # Recurse on children, update alpha/beta and prune if we see 'alpha >= beta'.
10    best_moves, num_evals = [], 0
11    for s in state.generate_next_states():
12        child_moves, child_score, child_evals = minimax_search_alphabeta(s,
13        alpha, beta, heuristic_fn,
14        depth_limit - 1, not maximize)
15        # Found better child, needs update.
16        if maximize and child_score > alpha:
17            alpha = child_score
18            best_moves = child_moves
19        elif not maximize and child_score < beta:
20            beta = child_score
21            best_moves = child_moves
22        # Increment number of evaluations.
23        num_evals += child_evals
24        # Prune the rest branches because this whole subtree is not active.
25        if alpha >= beta:
26            break
27    # Enqueue current state to the moves.
28    return ([state] + best_moves, alpha if maximize else beta, num_evals)

```

Progressive Deepening

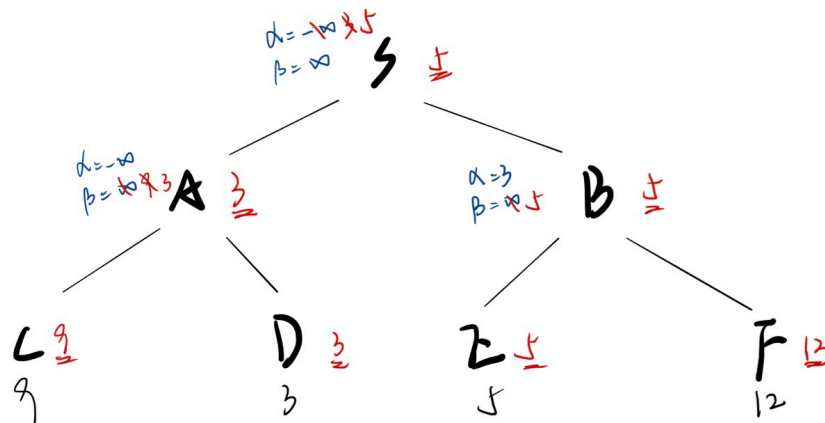
Still, real-life games may require strict timing at each turn. **Progressive Deepening** is a kind of *Anytime Algorithm*, where we do α, β on depth limit from $1 \rightarrow \infty$ (well, actually maximal depth of this game). After each α, β , we update the choice. Whenever time is up, we return the current decided choice. Since searching on a shallow depth is fast, it is guaranteed to give a solution. Longer time limits give us chance to do deeper searches and get a better choice.

When using progressive deepening, we can also deploy the **reordering** trick to increase the chance of pruning at α, β for later iterations. Bottom up, at a MAX layer, sort children in *non-increasing* order from left to right; vice versa. Then, at the following deeper α, β searching, the right branch will more likely to be pruned.

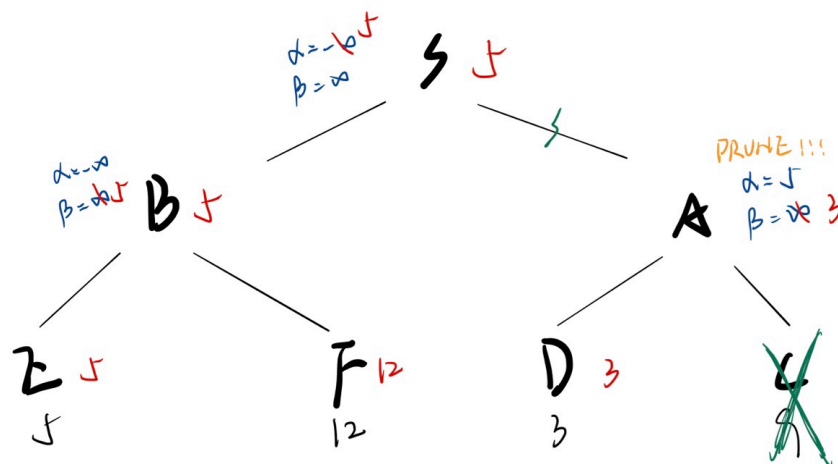
Example

Example from [2015 Quiz 1](#) & [2017 Quiz 1](#):

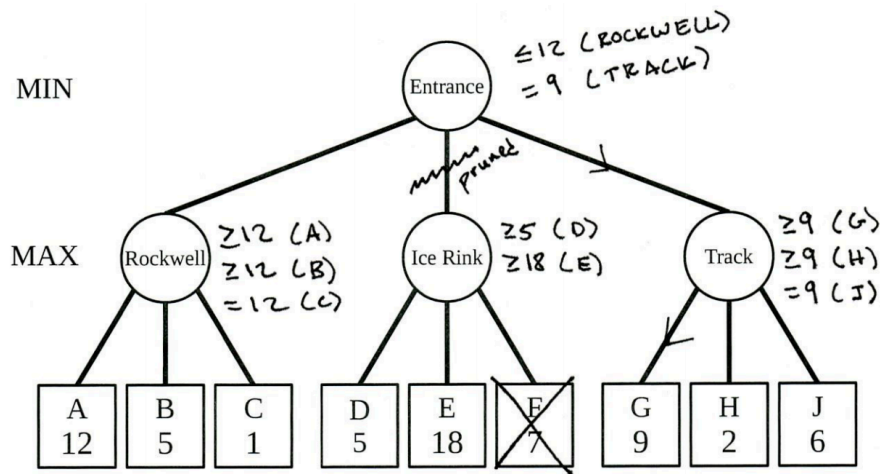
The minimax with α, β pruning on the following tree [top layer MAX, second layer MIN, bottom layer endgames], NO pruning happens:



The minimax with α, β pruning with a better ordering, pruning happens (C does not actually get statically evaluated):



Another clear example with $\leq, \geq, =$ notations:



Constraints Solving Problems

A **Constraints Solving Problem** (CSP) is a problem consisting of:

- **Variables**
- **Values** - each variable v can be assigned one value x / currently unassigned (have no assigned value yet)
- **Domains** - each variable v has a domain D of values that can be assigned to, may get updated during the search for a solution
- **Constraints** - a constraint $C(v_1, v_2)$ specifies a condition to be satisfied between v_1 's value and v_2 's value
 - If there is at least one constraint between v_a and v_b , we say they are *neighbors*

The goal is to *assign* each variable to a proper value in its domain with all the constraints satisfied (possibly not solution or multiple solutions).

Example:

```

1 Variables = {A, B, C}
2 Initial_domains = {
3     A: [1, 2],
4     B: [0, 1],
5     C: [0, 1, 2]
6 }
7 Constraints = [
8     "A != B",
9     "A != C",
10    "C > A",
11    "C > B",
12    "B + C < 3"
13 ]

```

One solution to this example is $A = 1, B = 0, C = 2$.

Depth-First Solver

A naive algorithm to a CSP is to apply DFS on the search tree w/o any reduction (i.e., check only the assigned variable):

```

1 def constraints_solving_dfs(problem):
2     # Initial agenda contains the original snapshot.
3     agenda, extension_count = [problem], 0
4     while len(agenda) > 0:
5         csp = agenda.pop()
6         extension_count += 1
7         # Some variable's domain reduced to empty / Some constraints are
violated, then BACKTRACK.
8         if has_empty_domains(csp) or not check_all_constraints(csp):
9             continue
10        # All values assigned, then got a solution.
11        if len(csp.unassigned_vars) == 0:
12            return (csp.assignments, extension_count)
13        # Add the new problems with the possible values assigned for the next
variable to agenda.
14        var = csp.pop_next_unassigned_var()
15        for val in csp.get_domain(var)[::-1]:
16            agenda.append(csp.copy().set_assignment(var, val))
17        # None of the assignment combinations work, so no solution.
18        return (None, extension_count)

```

This will result in a huge number of meaningless extensions during the search.

Optimizations with Reductions

To improve, we consider doing different levels of reductions. Whenever we give a new assignment and want to add that into the agenda, we propagate through the graph and try to *eliminate* some values from some variables' domains that are certainly impossible.

```

1 def propagate(enqueue_condition_fn, csp, var) :
2     agenda = [var]      # This is the propagation agenda, not the CSP solver's
agenda.
3     while len(agenda) > 0:
4         var = agenda.pop(0)
5         # Iterate through all neighbors.
6         for neighbor in csp.get_neighbors(var):
7             # If given a value w in the neighbor's domain, for all current
variable's values v,
8             # some constraint is violated, then w is certainly not possible to be
assigned to
9             # the neighbor. Eliminate w from the neighbor's domain.
10            to_del = set()
11            for v_neighbor in csp.get_domain(neighbor):
12                violates_fn = lambda v_var: any(map(lambda c: not c.check(v_var,
v_neighbor),
13                                                    csp.constraints_between(var,
neighbor)))
14                if all(map(violates_fn, csp.get_domain(var))):
15                    to_del.add(v_neighbor)
16            # This neighbor's domain gets reduced.
17            if len(to_del) > 0:
18                for v_neighbor in to_del:
19                    csp.eliminate(neighbor, v_neighbor)
20            # Check how far still to propagate.

```

```

21         if enqueue_condition_fn(csp, neighbor) and neighbor not in
agenda:
22             agenda.append(neighbor)
23
24 def constraints_solving_generic(problem, enqueue_condition) :
25     ...
26     var = csp.pop_next_unassigned_var()
27     for val in csp.get_domain(var)[::-1]:
28         agenda.append(csp.copy().set_assignment(var, val))
29         propagate(enqueue_condition, agenda[-1], var) # Propagation happens
here.
30     ...

```

To distinguish between different strategies of propagation, we have different enqueue condition (whether to propagate further or not):

- **Forward-Checking** (only check current var's neighbors, i.e., no propagation): enqueue condition is always `False`
- **Domain Reduction / Propagation through Reduced Domains** (whenever a neighbor n 's domain is reduced, continue on n 's neighbors): enqueue condition is always `True`
 - Doing so after every assignment is very expensive, but it can do elimination the most thoroughly
 - Given an initial problem, we can also do this once before the search (setting initial propagation agenda to all variables in the problem) → This simplifies the problem later to be solved
- **Propagation through Singleton Domains** (continue on a neighbor n 's neighbor only if n now has only one possible value in its domain): enqueue condition is `csp.get_domain(n)` has only 1 element
 - This sacrifices some eliminations but can be done much faster than propagating through all reduced domains

Example

Example from [2014 Quiz 2](#): omitted.

- Do assignment → list constraints involving the assigned variable in this step → reduce & update the domains (globally) → propagate following the desired rule
- Order matters: most constrained (most edges) first

Learning Methods

Learning by Nearest Neighbors

When handling numeric data, map data onto a n -dimensional feature space, classify each input by its nearest neighbor target. Classification is based on which of the target point (point with known classification) is the *nearest* to a given data point. To avoid calculation of all distances for each input data point, we typically pre-calculate the **Decision Boundary**, and then only use these boundaries to classify input data points.

Different definition of distance (suppose mapping onto a *Cartesian* space):

- **Euclidean**: $Euclidean(\vec{u}, \vec{v}) = \sqrt{\sum (u_i - v_i)^2}$
- **Manhattan**: $Manhattan(\vec{u}, \vec{v}) = \sum |u_i - v_i|$

- **Hamming:** $Hamming(\vec{u}, \vec{v}) = \sum_{1 \leq i \leq n, u_i \neq v_i} 1$ (normally for binary / boolean coded vectors)
- **Cosine:** $Cosine(\vec{u}, \vec{v}) = 1 - \frac{\vec{u} \cdot \vec{v}}{||\vec{u}|| ||\vec{v}||}$ (1 minus cosine of the angle between two vectors starting from origin (0 ~ 2)), i.e., differentiating the ratio between different factor values

Difficulties:

- Proper extraction of features
- Narrow spread of values → Normalization, use "how much standard deviations is it away from the mean"

K-Nearest Neighbors: pick k nearest neighbors and they vote for a classification → pick the majority choice.

Learning by Building Identification Trees

When handling symbolic data:

- Non-numeric
- Some factors don't matter, some factors only conditionally matter

We can use an **Identification Tree** (ID Tree) to do such classification. Consider the vampire test problem, each data point has the following features: `Shadow: {true, false, unknown}, Garlic: {true, false}, Complexion: {pale, ruddy, avg}, Accent: {heavy, odd, none}, Vampire: {yes, no}`. Our goal is to use the previous four features to identify the last feature - a datapoint is vampire / not.

Calculation of a set's **disorder** from the Information Theory: $D(set) = -\frac{p}{t} \log_2 \frac{p}{t} - \frac{n}{t} \log_2 \frac{n}{t}$, where p is the number of positive samples in the set, n is the number of negative samples in the set, and t is the total number $= p + n$. Disorder D is a value between 0 (least disordered, i.e., homogeneous) → 1 (most disordered, i.e., half-half). Here, if the target feature is non-binary, $D(set)$ can be defined as

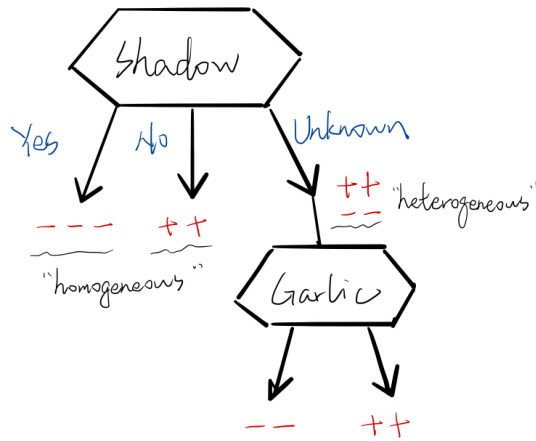
$$D(set) = \sum_{\text{classification } c} -\frac{n_c}{n} \log_2 \frac{n_c}{n}.$$

A feature's overall disorderness can then be given by a weighted sum of branch:

$$\sum_{\text{branch } b} \frac{\# \text{ samples in } b}{\# \text{ all samples}} D(b's \text{ set}).$$

Pick the lowest-scored feature as the classifier of the current node.

If a branch is **homogeneous**, we are done on that branch. For each **heterogeneous** branch, we still need to inspect another feature to further separate those data apart. Repeat the feature selection process on the remaining features on data of that branch. Finally we will get an *Identification Tree*, where leaf nodes are all homogeneous (providing a classification of the target feature):



After optimizing, the logic behind this tree can be simplified as:

```

1  if shadow or not garlic:
2      not vampire
3  else:
4      is vampire

```

The principle of **Occam's Razor**: law of parsimony, entities should not be more complicated without necessity. A simpler ID tree is more likely to be correct for general data, while a sophisticated ID tree probably *overfits* the training data.

Example

Exmample from [2015 Quiz 2](#): omitted.

- In a greedy *disorder-minimizing ID tree*, the same test will NOT appear twice in the same branch.
- In a greedy *disorder-minimizing ID tree*, should never use a test with only one branch (does nothing), unless no other tests can do better and breaking the tie gives that one-branch test.

Basic Neural Nets

Neural nets are biologically inspired classification model.

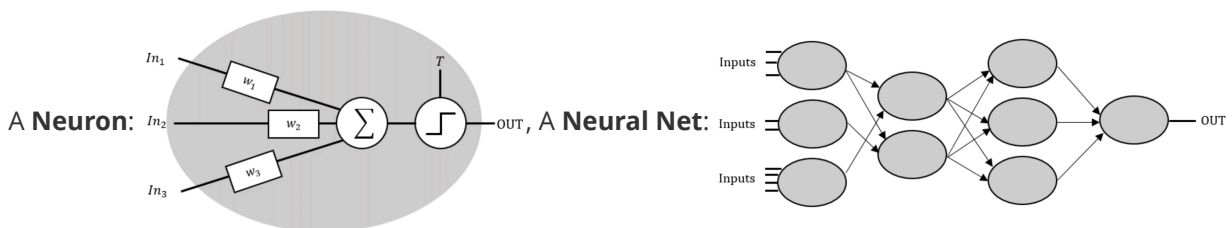


Figure from 6.034 lab 6. (T defaults to -1)

Boolean Logical Neural Nets

When input/output are all boolean (0/1), a combination of neurons can be used as logical gates. For example, a neuron with 2 inputs, both with weight 1, and a threshold $T = 1.5$, is an **AND** gate; a threshold $T = 0.5$, is an **OR** gate.

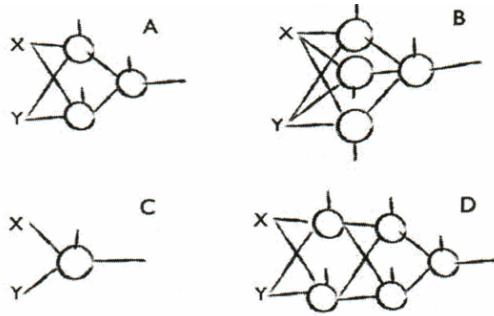
Visualization guideline:

1. On *input layer*, every boundary line needs a neuron, which can represent the sub plane on one side of this line









- If can represent the region, done;
 - Else, need following logical gates layers to combine these sub planes
- Find out the logical combination expression of these sub planes, and use corresponding neurons
 - For minimal such neural nets, need some tricks...
 - $A \text{ AND } B / A \text{ OR } B$: [2, 1]
 - $A \text{ AND } B \text{ AND } C$: [3, 1]
 - $(A \text{ AND } B) \text{ OR } (A \text{ AND } B)$: [2, 2, 1]
 - $(A \text{ AND } B) \text{ OR } C$ can be done in [3, 1]

Example from [2013 Quiz 3](#):

Given these four neural nets



The following pattern can be represented by:

			
ABD	$NONE$	$NONE$	D
			
$ABCD$	B	B	ABD

Activation Functions

Typically used **Activation Functions**:

- Stairstep (i.e., Threshold): $y = 1$ if $x \geq T$, otherwise 0
- Sigmoid: $y = \frac{1}{1+e^{-S(x-M)}}$; when $S = 1, M = 0, \frac{dy}{dx} = y(1 - y)$
- ReLU: $y = \max(0, x)$

Performance Measurement

The most basic performance measurement is $P = -\frac{1}{2}(\text{target} - \text{out})^2$. Its derivative is $\frac{dP}{dout} = \text{target} - \text{out}$.

Forward Propagation

From input layer \rightarrow output layer, calculate neuron outputs until we get the final output of this neural net.
Too simple, just omitted here.

Backward Propagation

The goal of **Backward Propagation** is to derive the partial derivative $\frac{\partial P}{\partial w_{A \rightarrow B}}$ for each weight, thus we know how to adjust this weight value to improve the performance.

This gradient calculation has a reusable part δ for each neuron, that can be reused for all neurons in its previous layers. This shows the *dependencies* given by the chain rules. So, typically we first calculate the δ values for neurons:

$$\delta_B = \begin{cases} out_B(1 - out_B)(target - out_B), & \text{if B in output layer,} \\ out_B(1 - out_B) \sum_{successor C_i} w_{B \rightarrow C_i} \delta_{C_i}, & \text{otherwise.} \end{cases}$$

Then, the gradient is simply $\frac{\partial P}{\partial w_{A \rightarrow B}} = out_A \delta_B$.

And then apply linear **Gradient Descent** with *learning rate* r :

$$w_{A \rightarrow B} = w_{A \rightarrow B} + \Delta w_{A \rightarrow B} = w_{A \rightarrow B} + r \frac{\partial P}{\partial w_{A \rightarrow B}}.$$

We can then **Train** the neural net's weight values by repeating [forward + backward] epoches on training data until the performance is good enough.

Learning rate r tradeoff:

- Large: jumping around, never finding the maximum
- Small: stuck in local maximum

Support Vector Machines (SVMs)

SVMs are another classification model which tries to find a maximum separation between $+/ -$ samples. Maximization helps avoid *overfitting* (but still vulnerable).

SVM terminology:

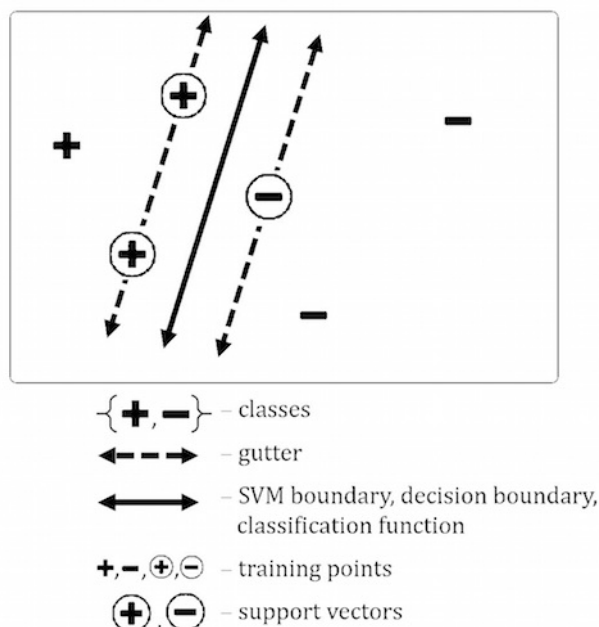


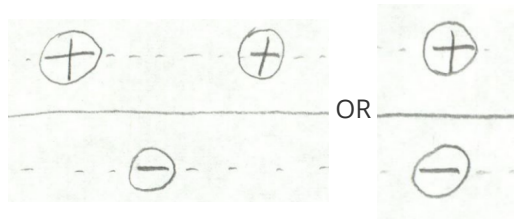
Figure from 6.034 lab 7.

- Core formula for an SVM is $\vec{w} \cdot \vec{x} + b$, where \vec{w} is the *weight vector*, b is *scalar offset*, and \vec{x} is a datapoint
- Not all vectors (points) on the *gutter* are necessarily *support vectors* - there might be ambiguity

Drawing Decision Boundary

For 2-D $+/-$ datasets:

1. Draw the *convex hull* for all $+$ points and $-$ points (contracting a rubber band around them)
2. Find where these two hulls are the closest, which can only be one of the following two situations:



Figures from [Roberts's note](#).

Mathematical Constraints & Calculations

Every SVM follows these constraints:

- $\vec{w} \cdot \vec{x} + b = 0$ defines the *decision boundary*
- $\vec{w} \cdot \vec{x} + b = \pm 1$ defines the positive/negative *gutters*
- Distance between two gutters $= \frac{2}{\|\vec{w}\|}$
- For all non-support vectors, $\alpha = 0$; For all support vectors, $\alpha > 0$, where α is the "supportiveness"
- $\sum_i y_i \alpha_i = 0$, where $y_i = \pm 1$ for positive/negative respectively
- $\sum_i y_i \alpha_i \vec{x}_i = \vec{w}$, (a higher- α SVM's gutter is closer to the boundary)

The process of determining \vec{w}, b for Quiz questions (not training, because we find the boundary by hand first and calculate the parameters. In actual training of SVMs, start from a bunch of initial α values, and optimize on maximizing the gutter width):

1. Find the boundary line $mx + ny = k$ and write it in the way $\begin{bmatrix} m \\ n \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + (-k) = 0$
2. Use Eqn-2., pick a positive support vector \vec{x} , which gives $\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \cdot \vec{x} + b = 1 \Rightarrow$ scale the parameters above to get actual \vec{w} and b
(\vec{w} should be perpendicular to the boundary, pointing towards $+$ region)

Finding α values: Solve equations given by Eqn-5. and Eqn-6.,

$$\begin{cases} \sum_+ \alpha_i = \sum_- \alpha_i \\ \sum_+ \alpha_i \vec{x}_i - \sum_- \alpha_i \vec{x}_i = \vec{w} \end{cases}$$

NOTE: On a two-supports gutter, moving one support vector A closer to the perpendicular point makes its $\alpha_A \uparrow$ and the other B's $\alpha_B \downarrow$.

Transformation & Kernels

There will be situations that such simple SVMs cannot handle. Here we need feature transformations to transform the original data into a different representation where they can be correctly separated.

Doing so for every datapoint is tedious. Instead, we can change the **Kernel** which defines a way of computing the "dot product" of two vectors. Simple linear kernel $\vec{w} \cdot \vec{x}$ can only handle a straight line boundary. Fancier kernels can define curves and thus give better classification for more complicated datasets:

- Quadratic kernel $(\vec{w} \cdot \vec{x} + 1)^2$ can draw quadratic curves
- Polynomial kernel $(\vec{w} \cdot \vec{x} + 1)^n$ can draw higher dimensional polynomial curves
- Radial-basis kernels can transform the 2-D dot product into infinite dimensional spaces, so can give very sophisticated separation but will emphasize overfitting

Given a feature representation transformation $\phi(\vec{x}) = \vec{x}'$, the corresponding kernel is $K(\vec{u}, \vec{v}) = \vec{u}' \cdot \vec{v}'$.

If one $+$ and one $-$ sit on exactly the same point, no kernels can separate them then.

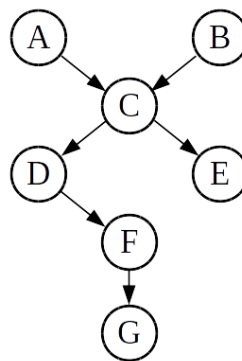
Bayes Nets

Independence of events (variables):

- (Marginally) Independent: $P(A|B) = P(A)$
- Conditionally independent: $P(A|BC) = P(A|C)$

Bayesian Network

A **Bayesian Network** (*Bayes Net*) encodes dependence between events. A variable is dependent on its parents:



Figures from 6.034 Lab 8.

where each variable is accompanied with a probability table, recording "given its parents, the probabilities of this variable taking certain values":

A	B	P(C=c1 A,B)	P(C=c2 A,B)
a1	b1	#	#
a1	b2	#	#
a1	b3	#	#
a2	b1	#	#
a2	b2	#	#
a2	b3	#	#
a3	b1	#	#
a3	b2	#	#
a3	b3	#	#

Figures from 6.034 Lab 8.

Suppose C can take values c_1, c_2, c_3 , minimum number of probabilities we need to store in C 's probability table above is $9 \times 2 = 18$. The c_3 column is not needed, because it can be inferred by previous columns as they add up to 1 for each row.

The *Bayes net assumption* says: Every variable is conditionally independent of its non-descendants, given its parents. For example, given C , D is independent of A and E .

Probability & Hypothesis

A probability of a hypothesis is in the form $P(\text{hypothesis} | \text{givens})$. For a Bayes net with variables A, B, C, D, E :

- **Joint** probability: $P(ABCDE)$
 - Can be computed with a chain rule from the top of the net to the bottom
- **Marginal** probability: e.g., $P(CDE)$, $P(A)$
 - Can be computed with a partial chain rule
 - Can be computed as a sum of joint probabilities, with remaining variables traversing all their possible values
- **Conditional** probability: e.g., $P(A|C)$ is a ratio of marginal probabilities
 - Can be computed as a ratio of marginal probabilities
 - Can be simplified if the givens contain all A 's parents but no A 's descendants. In this case, all A 's non-descendants can be removed from the givens

Their mathematical relationships are recorded here: [READ Flowchart](#).

Checking Independence

Two variables in a Bayes net can be **structurally independent** (refer to the definition of a Bayes net). A question like " $P(A|BDF) = P(A|DF)$?" can be converted to "Are A and B independent, given D, F ?". A question like " $P(A|BDF) = P(A|F)$?" can be converted to "Are A and B independent, given F ? \wedge Are A and D independent, given F ?".

Use *d-separation* algorithm as a formal procedure to determine structural independence:

1. Draw the *ancestral graph*, a subnet with only the variables mentioned and their ancestors
2. *Moralize*: for each pair of variables with a common child, draw an undirected line between them
3. *Disorient*: replace all directed edges with undirected lines
4. Delete all the givens (e.g., D and F) and removed their edges
5. Read the answer:
 - If A and B are disconnected, they are structurally independent
 - If one or both of them are missing, they are independent
 - If A and B are connected (have a path), they are not structurally independent

Furthermore, two variables A and B may not be structurally independent, but A happen to have approximately the same probability values in all table entries w/ or w/o B given. In this case, they are **numerically independent**. Structural independence is a special case of numerical independence.

Bayes net with all variables independent has no edges; with NO variables independent follows either a mesh or a chain.

Boosting

Boosting is a technique to combine a bunch weak classifiers into a strong *ensemble/amalgam* classifier which can correctly all or most samples.

Adaboost

Here we consider the **Adaboost** (*Adaptive Boosting*) algorithm on two classifications to get a strong classifier H :

1. Initialize all N training points with equal weights $w_i = \frac{1}{N}$
2. Compute the error rate E_h of each weak classifier h as the sum of weights of all points it misclassifies
3. Pick the "best" weak classifier h
 - Might be the one with the smallest error rate
 - Might be the one with error rate furthest from $\frac{1}{2}$
 - (Error rate close to 100% results in a strongly negative voting power)
4. Use the error rate of h to compute the voting power α of h , $\alpha_h = \frac{1}{2} \ln \frac{1-E_h}{E_h}$
5. Add h into the H : $H(x_i) = \text{sign}(\alpha_{h_1} h_1(x_i) + \dots + \alpha_{h_s} h_s(x_i))$ after round s
6. Update weights of each training point p
 - If p is classified correctly by h : $w_p \leftarrow \frac{1}{2} \frac{w_p}{1-E_h}$, going down
 - If p is misclassified by h : $w_p \leftarrow \frac{1}{2} \frac{w_p}{E_h}$, going up
 - (This makes h 's error rate now = $\frac{1}{2}$)
7. Repeat steps.2-7 until:
 - No good classifier remains (all have error rate $\frac{1}{2}$)
 - Reached max number of iterations
 - H is good enough (H misclassifies few points)

This gives us an ensemble classifier $H(x_i) = \text{sign}(\alpha_{h_1} h_1(x_i) + \dots)$. Weak classifiers may be selected multiple times. H 's error rate oscillates but is exponentially bounded.

Spiritual/Right-now Notes

Quiz 2 SRN

1. **Prof. Howard Shrobe** - Constraints in interpretation:
 - Constraint propagation in line-drawing analysis demonstrates that better description yields more constraint and faster performance.
 - Constraint exploitation in object recognition involves determining vertex locations using a small number of standard views.
 - Drawing-understanding program is most dependent on using constraint propagation to find consistent interpretations.
 - Drawing-understanding program is best attributed to joint efforts of people with diverse characteristics.
2. **Dr. Marc Raibert** - Boston Dynamics
 - Exhibited robots: Spot (dog), Handle (chicken), Atlas (human).
3. **Prof. Ed Boyden** - Biological engineering
 - A *core motivation* of his research is to build better tools to study the brain.
 - Space & Time: difference in scale between components of the brain and the quick neurological impulses compared to other biological processes, makes it *difficult* to study and model brain.
 - MAP: Expansion Microscopy

- Describe a procedure that enlarges brains by infusing diaper polymers that swell.
- CONTROL: Optogenetics
 - Describe how rats can be respond to light without normal retinal photoreceptors.
 - Optogenetics has to do with optical stimulation of genetically altered neurons.
 - Light can be used as a reward mechanism because fiber optic implants can be directed toward genetically engineered photoreceptor neurons.
 - (Describe a procesure involving direct electrical stimulation of brain tissue so as to stop hand tremors.)
- OBSERVE: Fluorescent Imaging
 - His researchers could more clearly image presynaptic and postsynaptic connection sites.
- 4. **Prof. Peter Szolovits** - Deep neural nets
 - Performance measurement → Activation → Training (partial deriv by chain rules) → Threshold (by bias) → Set learning rate → Re-use computation to save complexity.
 - *Pooling* refers to a mechanism that reduces input size by passing along only one value (e.g., max value) from each group.
 - Sudden *success* of deep neural nets is best attributed to increases in computing speed.
- 5. **Prof. Aleksander Madry** - Adversarial attacks

Quiz 3 SRN

1. **Prof. Howard Shrobe** - AlphaGo:
 - Bootstraps up from random behavior by playing itself.
 - Uses limited-depth rollouts.
2. **Prof. Peter Szolovits** - Learning in sparse spaces: Learning phonological rules using Sussman-Yip:
 - Is possible because negative examples prevent overgeneralization (Stops generalizing when an evolving pattern matches a negative example).
 - Starts from a "seed" example which it generalizes.
 - Forms its own negative examples by exploiting the fact that words have only one plural form.
 - Uses several examples to learn pluralization rules.
 - Sparseness of phonemes in a high-dimensional distinctive-feature space.
3. **Dr. Kimberle Koile** - Genetic algorithms:
 - Can be viewed as a kind of hill climbing.
 - Involve a mechanism that helps to avoid problems with local maxima.
 - Use *mutations* for local optimizations and *crossovers* for global jumps.
 - Works better if diversity is a component of fitness determination.
 - Karl Sim's creatures are so successful because the domain had many solutions, allowing genetics to produce a variety of outcomes.
4. **Prof. Randall Davis** - Represetations:
 - Transition space suggests that in human thinking, change causes change.
 - Transition space works because a small vocabulary of change descriptors can describe complicated events.
 - Architectures using chunking remember useful sequences of actions.
 - Case-frame (Role-frame) representation has slots for entities participating in events.
5. **Dr. Kimberle Koile** - Near-miss learning:
 - Makes it possible to learn something definite with every sample, positive or negative.
 - Positive examples leading to generalization & negative examples cause specialization.
 - Specializes knowledge through forbidding and requiring relation links.

- Number of samples approximating the number of characteristics learned.
- Uses a climb-tree heuristic when a positive sample has a different type from the evolving model.
- Starts with a seed example and then makes modifications to both limit and expand matches.

6. **Prof. Nancy Kanwisher** - Brain functional MRI (fMRI):

- Indicated we have special purpose brainware dedicated to processing images of body parts.
- Indicated we have special purpose brainware dedicated to words.

Quiz 4 SRN

1. **Prof. Randall Davis** - Architecture, General Problem Solver (GPS):

- Subsumption architecture features robust action by way of functional abstraction
- GPS is based on means-ends analysis

2. **Dr. Gill Pratt** - Toyota Research Institute:

- Human drivers are extremely competent when viewed from a fatalities-per-mile perspective
- Chauffeur is made for full autonomy and Guardian for driver assistance

3. **Dr. Boris Katz** - NLP, START:

- START suggested the feasibility of Watson and Siri
- Translates English into sets of ternary expressions
- A good way to collect information needed for constructing a large set of natural language questions is to gather semi-structured data from the information boxes on the top of Wikipedia pages

4. **Prof. Robert Berwick** - Language & Evolution, Merge:

- Merge is associated with completion of an anatomical loop in the brain
- An important outcome of the merge operator in humans is the ability to create hierarchical representations
- Merge is as a language CPU which processes speech into meaning and vice versa

5. **Dylan Holmes** - Story understanding, Genesis:

- Uses if-then rules to build a graph of causal connections
- Models different cultures with different sets of rules and concepts
- Key idea is the use of rules and search for reasoning and conceptual analysis
- Involves using a carefully tuned prior that is a function of model size to guide Bayesian structure search
- Tells stories persuasively by filtering content
- Reflects our human tendency to seek explanations
- The role of stories in human intelligence explains why merge matters

6. Bonus - The AI Biz: ~~None recorded.~~