The Nearest Neighbor Classifier

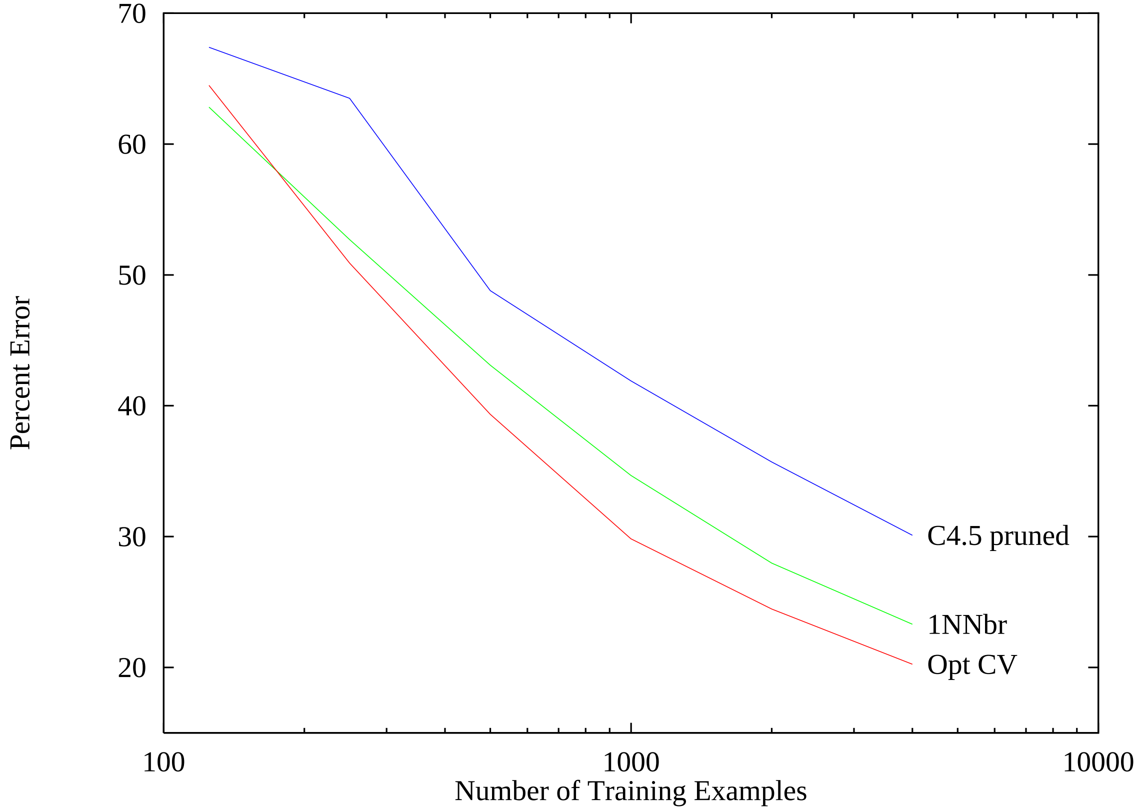
- **Store every training example:** $\langle \mathbf{x}, f(\mathbf{x}) \rangle$.
- **To classify a new example** x'**:**

  Find most similar previous example $\langle \mathbf{x}^*, f(\mathbf{x}^*) \rangle$.

  Guess $\hat{f}(\mathbf{x}') = f(\mathbf{x}^*)$.
- **Measure similarity by Euclidean distance**.

$$d(\mathbf{x}, \mathbf{x}') = \sum_{j=1}^{n} (x_j - x'_j)^2$$

The Nearest Neighbor Algorithm is Competitive



Nearest Neighbor Critically Depends on Distance Metric

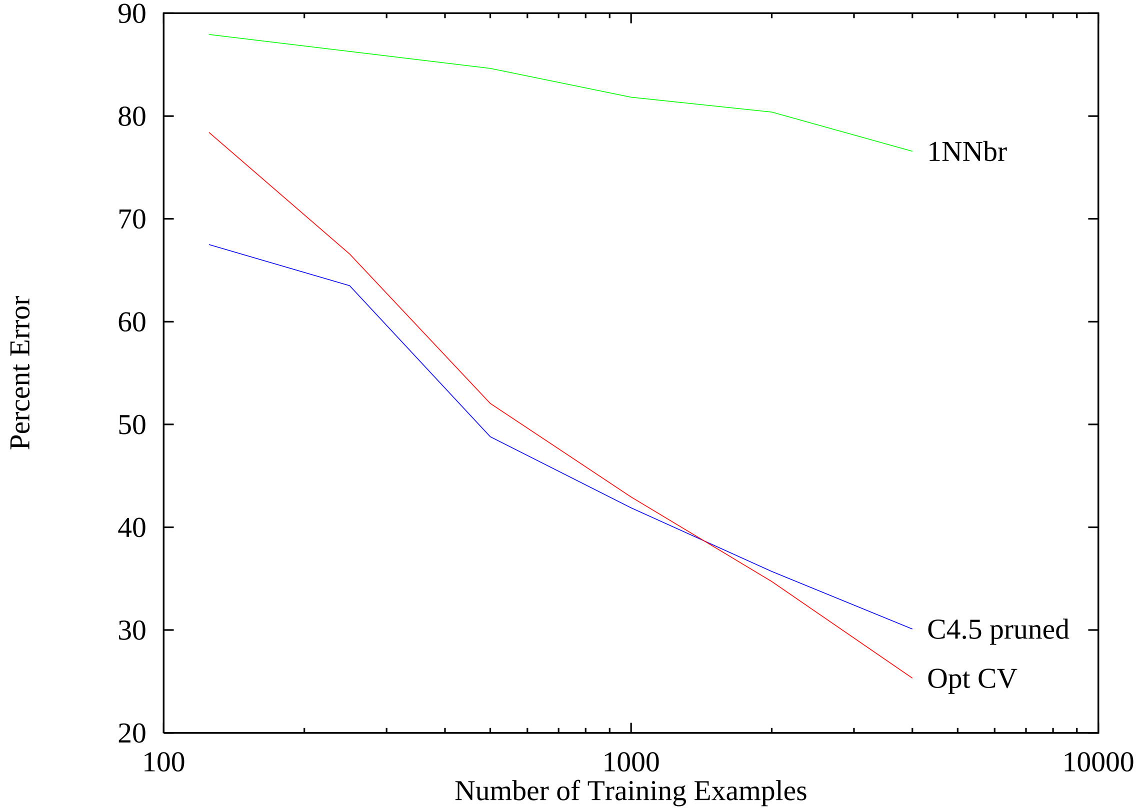
The key assumption of Nearest Neighbor is that similar cases have the same class.

- **Normalize Feature Values**. All features should have the same range of values (e.g., $[-1, +1]$). Otherwise, features with larger ranges will be treated as more important.
- **Remove Irrelevant Features**. Irrelevant or noisy features add random perturbations to the distance measure and hurt performance.
- **Learn a Distance Metric**. Weight each feature by its mutual information with the class.

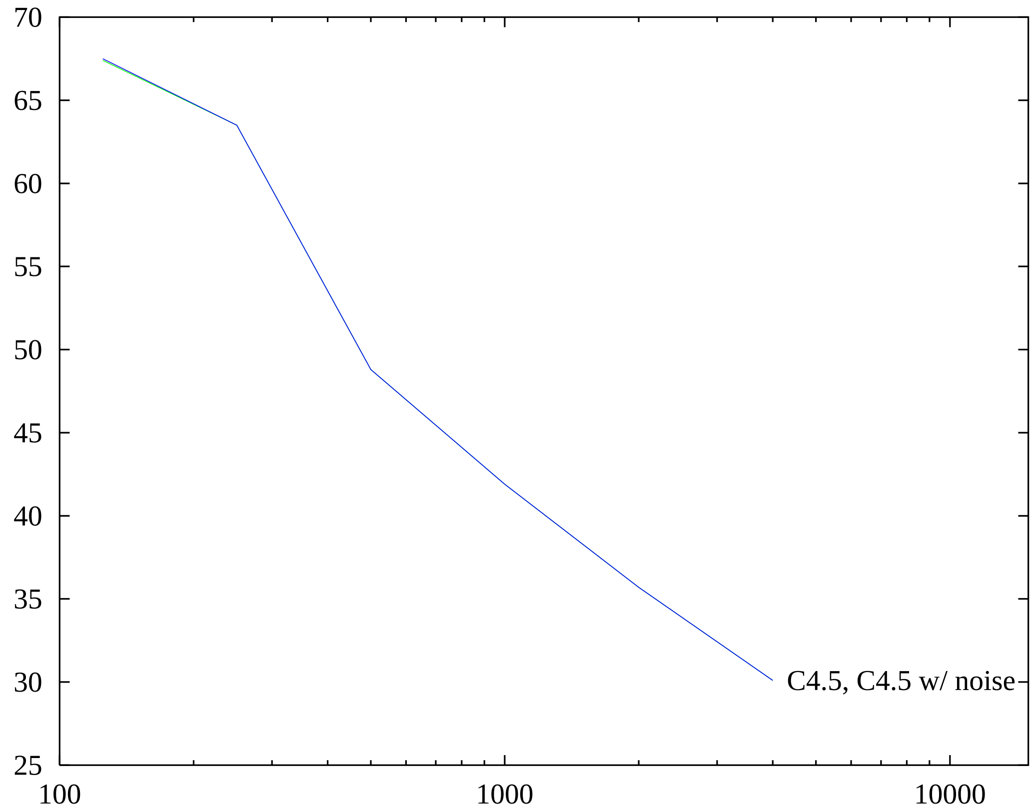
$$d(\mathbf{x}, \mathbf{x}') = \sum_{j=1}^{n} w_j (x_j - x'_j)^2$$

- **Smoothing**. Find the k nearest neighbors and have them vote. This is especially good for "class noise".

Nearest Neighbor with 10 Random Noise Features

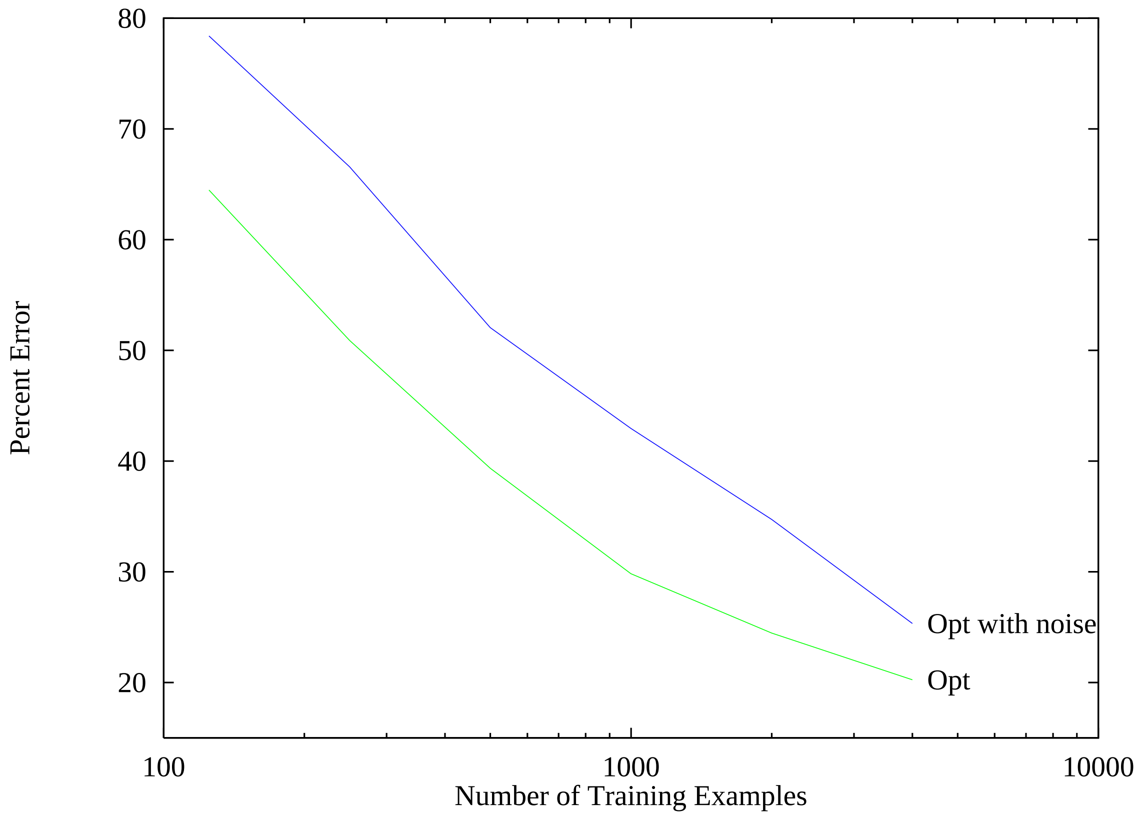


C4.5 With and Without Noise



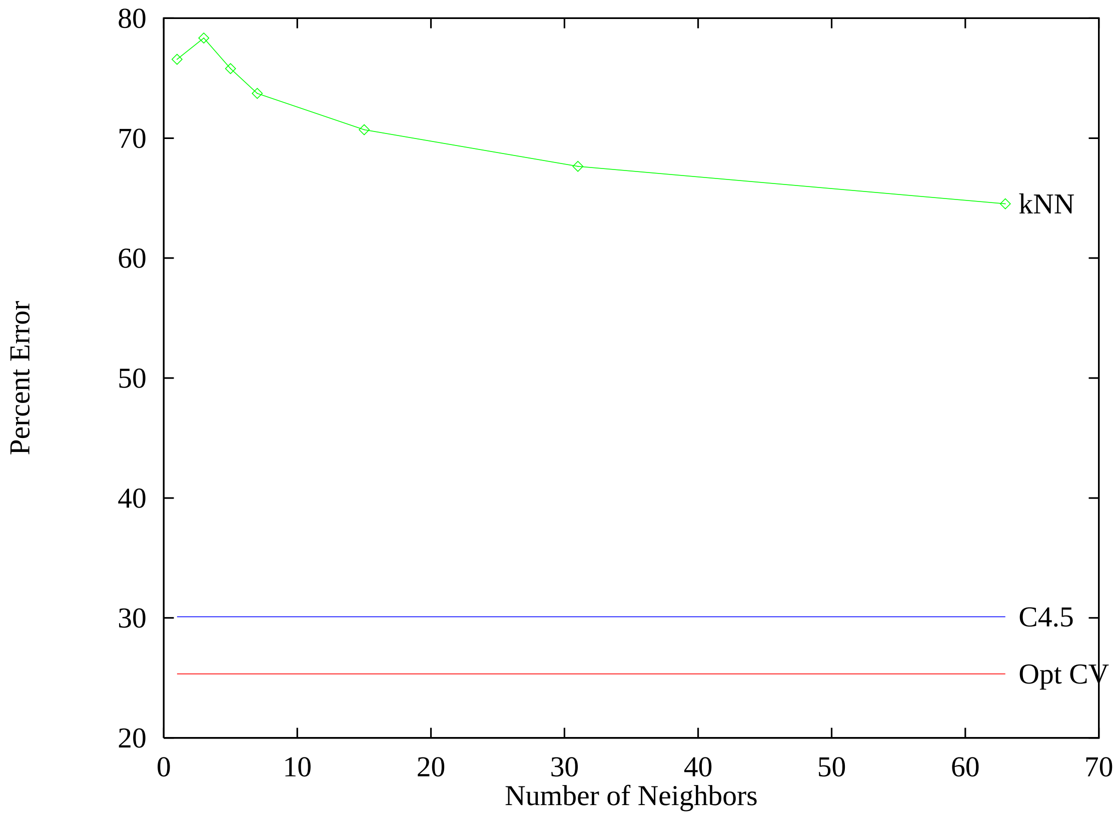
The mutual information heuristic is very good at identifying the 10 extra noisy features. The only difference in performance was for 125 examples.

Opt With and Without Noise



Backpropagation had more difficulty identifying the 10 extra noisy features. It clearly suffered, even with large sample sizes.

Smoothing By Using the k Nearest Neighbors

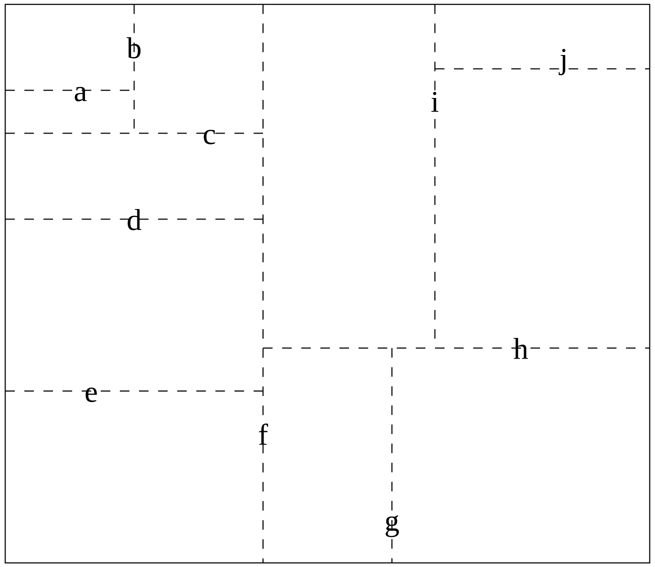


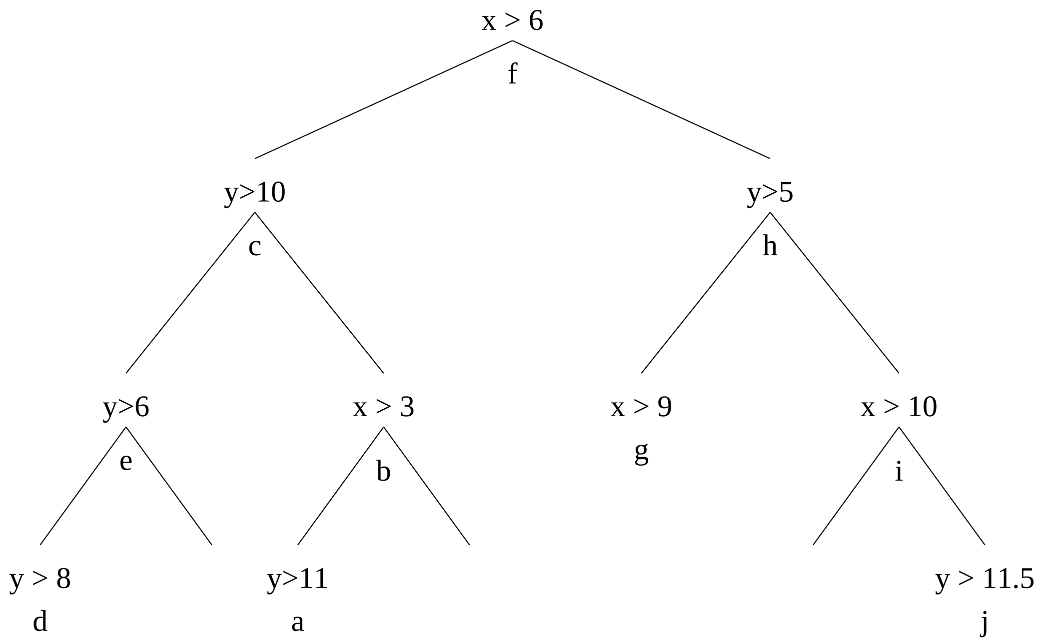
Training on 4000 examples. Large values of k give some improvement, but not much.

Reducing the Cost of Nearest Neighbor

- **Efficient Data Structures for Retrieval (kd-trees)**
- **Selectively Storing Data Points (editing)**

A kd-tree is like a decision tree, except that we split using the *median* value along the dimension having the highest variance.





Log-time Queries with kd-trees

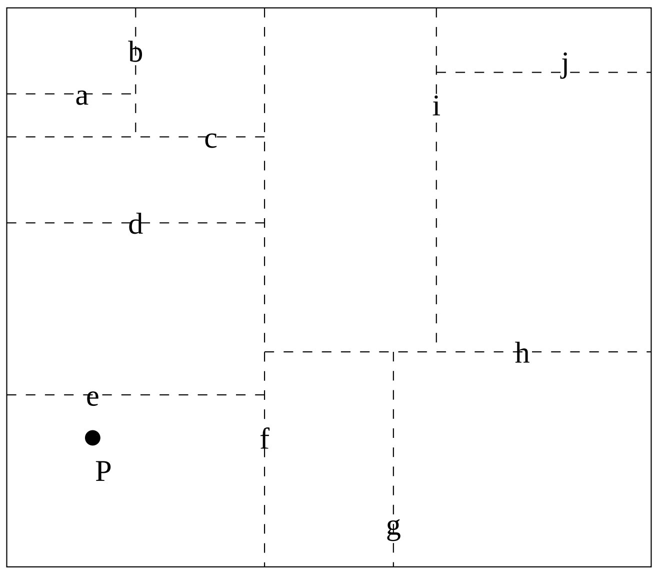
Given a query point P, find the nearest stored point.

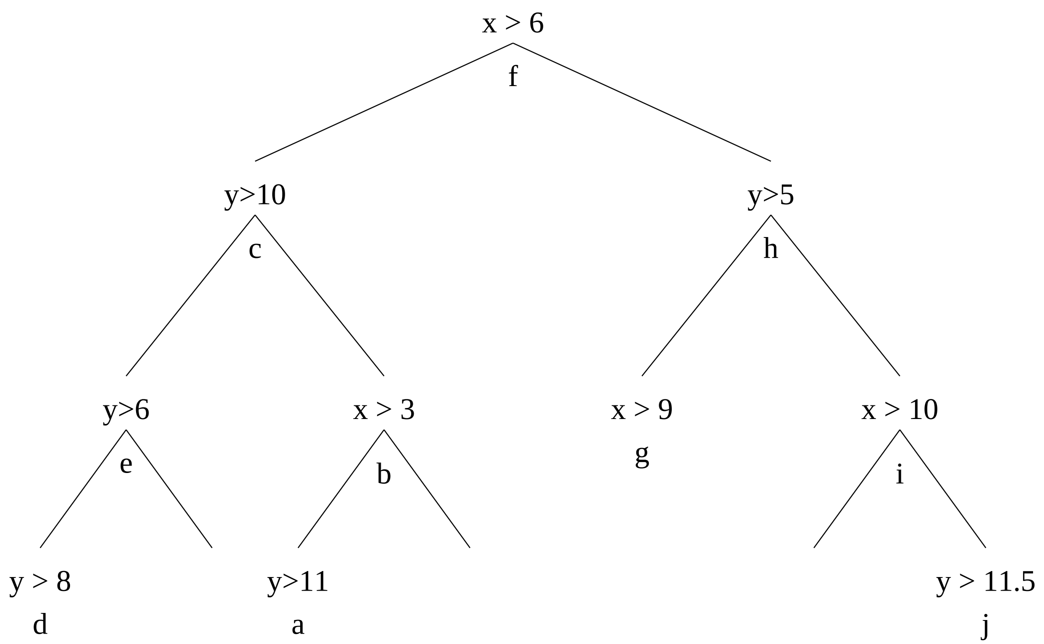
- **priority queue PQ of tree nodes that have not yet been searched, sorted by minimum distance to P**
- **nearest point found so far and its distance D**

Given the nearest node in the tree

- **does this node $= P$?**
  If so, return this node with distance of zero.
- **compute distance of this node to P**
  Is this the best distance so far? If yes, remember this point and its distance.
- **compute minimum distance from each child node to P**
  Push both of them onto the priority queue.
- **stop as soon as the nearest node on the priority queue has distance $\geq$ the best distance so far.**

Example





New Distance	Best distance	Best node	Priority Queue
none	∞	none	(f, 0)
4.00	4.00	f	(c, 0) (h, 4)
7.61	4.00	f	(e, 0) (h, 4) (b, 7.61)
1.00	1.00	e	(d, 1) (h, 4) (b, 7.61)

This is a form of A* search using the minimum distance to a node as an underestimate of the true distance.

Lookup Code

```
kdtree  root;
point nearestNeighbor(point P)
{
  heap<pair<node,float> > pq;        // priority queue
  float bestDistance = INFINITY;
  point bestPoint;
  pq.add(pair(root,0));
  while (!pq.isEmpty() && (bestdistance > pq.firstElement().second)) {
     pair<node,float> next = pq.pop();
     node  thisNode = next.first;
     float newDistance = distance(P, thisNode.point);
     if (newDistance == 0) return thisNode.point;
     if (newDistance < bestDistance) {
        bestPoint = thisNode.point;
        bestDistance = newDistance;
     }
```

Lookup Code (2)

```
    // now expand this thisNode to consider its children
    if (thisNode.test(P)) {
      pq.add(pair(thisNode.right,0));
      pq.add(pair(thisNode.left, P[thisNode.feature] − thisNode.threshold));
    }
    else {
      pq.add(pair(thisNode.left,0));
      pq.add(pair(thisNode.right, thisNode.threshold − P[thisNode.feature]));
    }
  }
  return bestPoint;
}
```

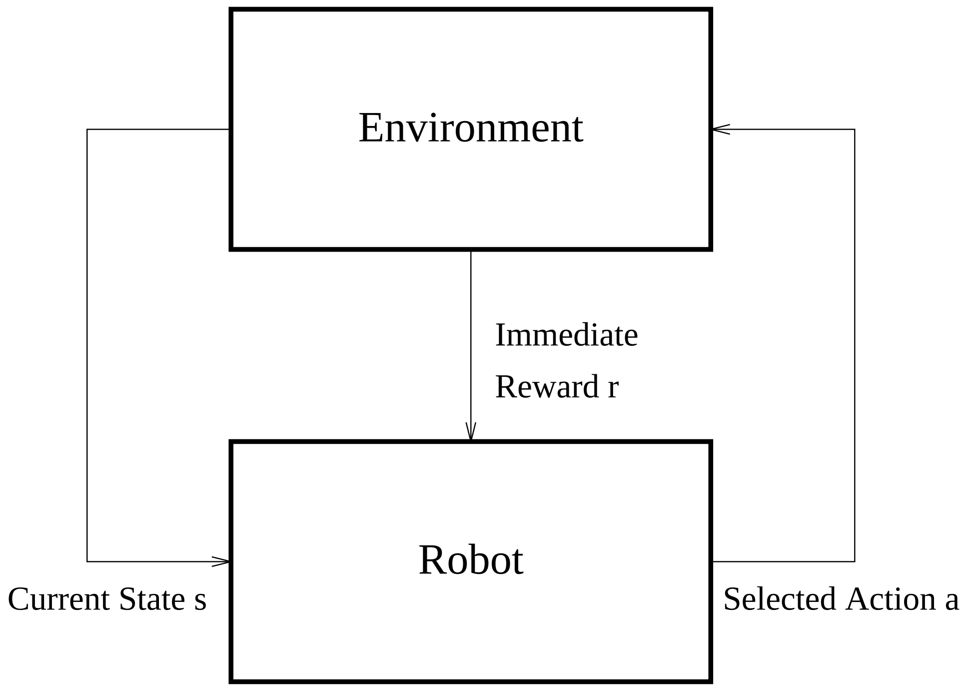
This algorithm can be modified to return the k nearest neighbors.

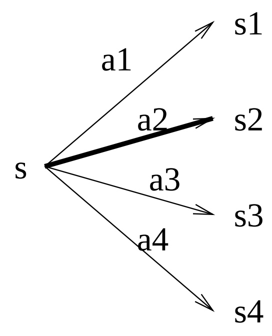
Edited Nearest Neighbor

Test each training data point to see if it can be correctly classified given the other points. If so, delete it from the data set and repeat.

Alternatively, only store a point if it is not correctly classified by the points already stored.

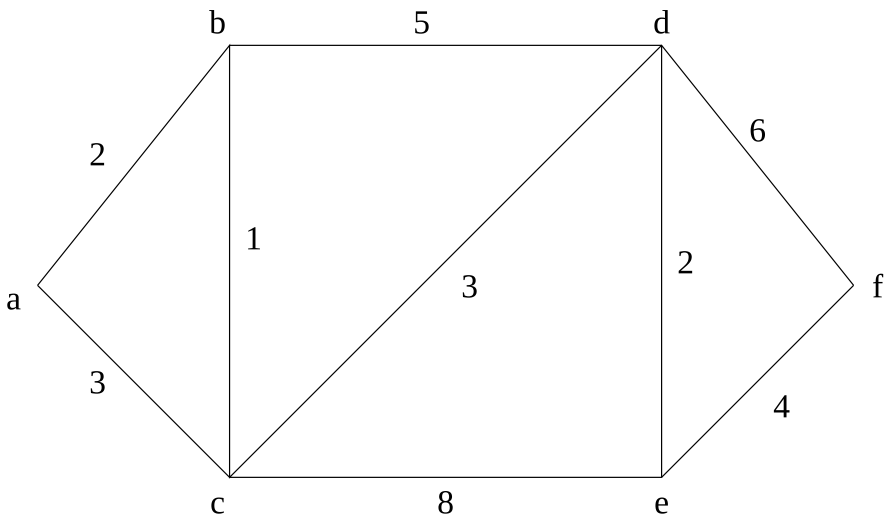
Learning in Search and Control: Reinforcement Learning





- **Robot: Sense state s, choose action $a = \pi(s)$.**
- **Environment provides scalar reward $r(s)$.**
- **Goal: Learn a policy π that maximizes total reward.**

Example: Searching in a Problem Space

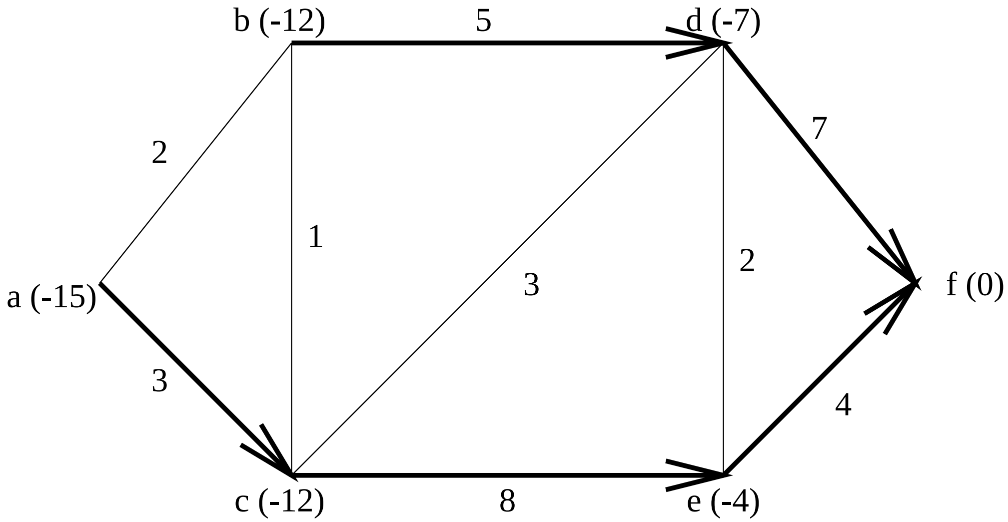


- **Start State: a**
- **Goal State: f**
- **Reinforcement:** $-cost$
- **Possible actions:** links leaving the state.

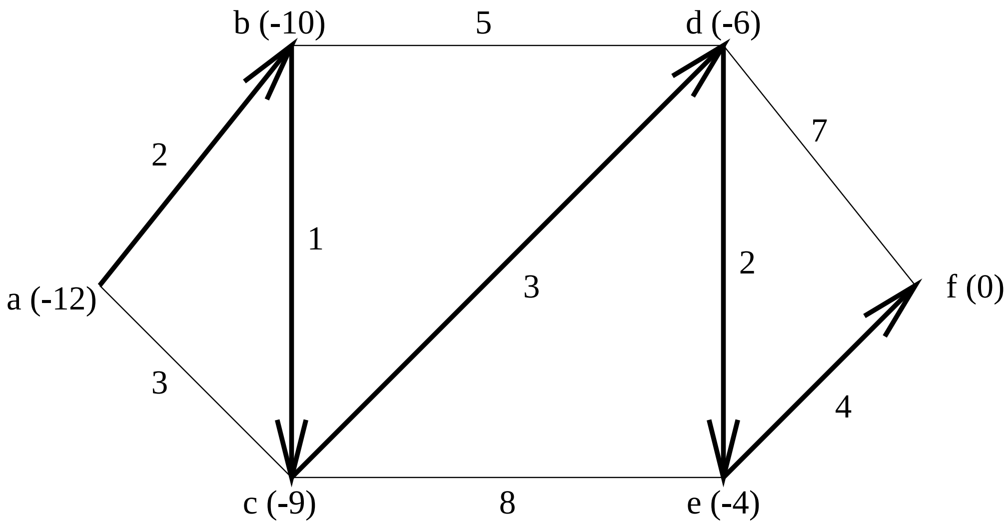
The Value Function for a Policy

- **Value function $f(s)$:** Total reward received when starting in state s and following policy π:

$$f(s) = r(\pi(s)) + r(\pi(\pi(s))) + r(\pi(\pi(\pi(s)))) + \ldots + r(s_{final}).$$



Optimal Value Function and Policy



- **If we know f, we can compute $\pi(s)$:**

$$\pi(s) = \operatorname*{argmax}_a \; r(a(s)) + f(a(s)).$$

- **This means that**

$$f(s) = \max_a \; r(a(s)) + f(a(s))$$

Value Iteration

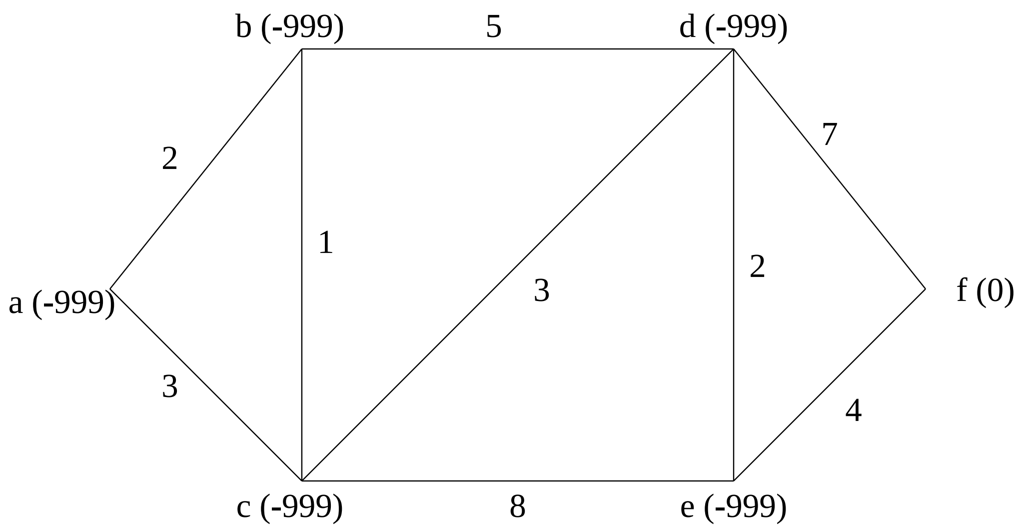
Start with $f(s) = -999$ for all states s.

Do forever:

  Choose s at random

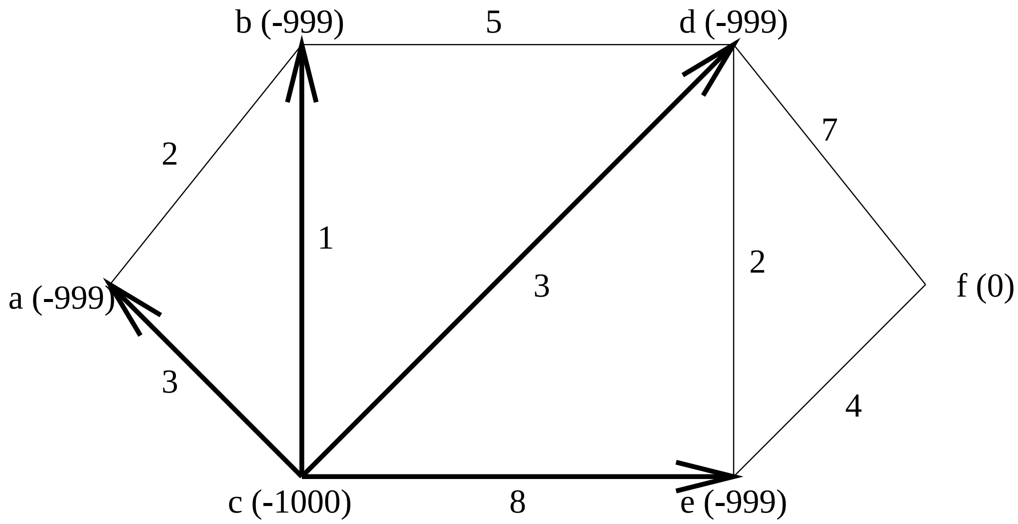
  Compute $BackedUpValue = \max_a \; r(a(s)) + f(a(s))$

  $f(s) := BackedUpValue$

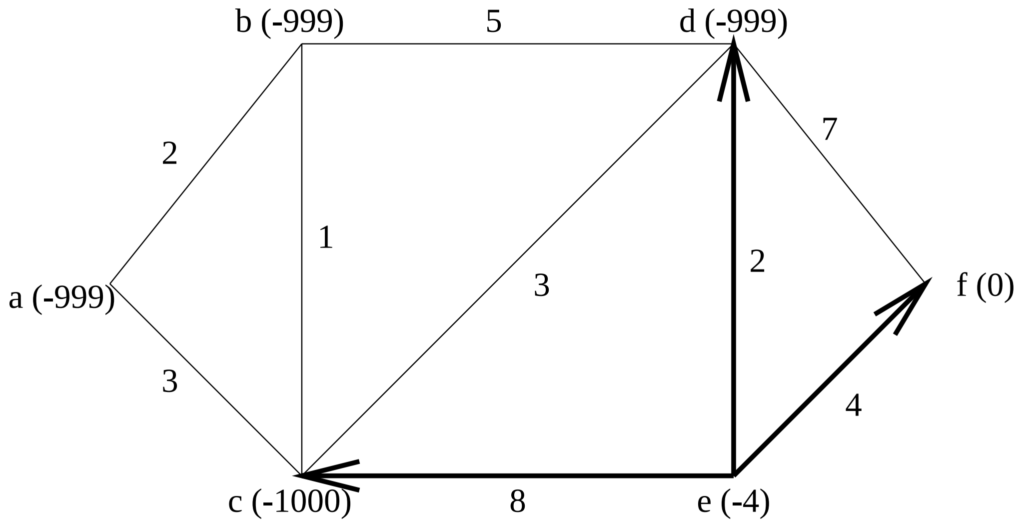


Value Iteration

Choose State **c**:

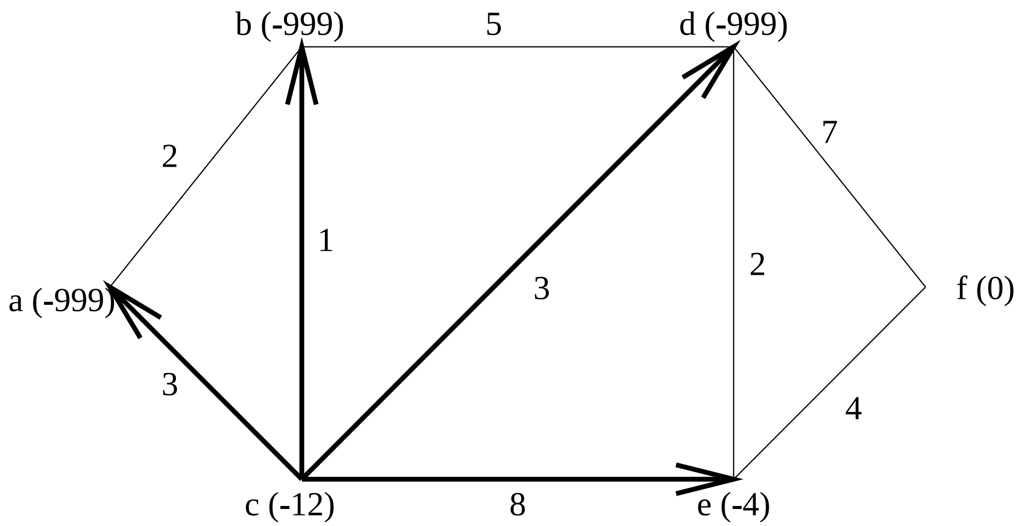


Choose State **e**:

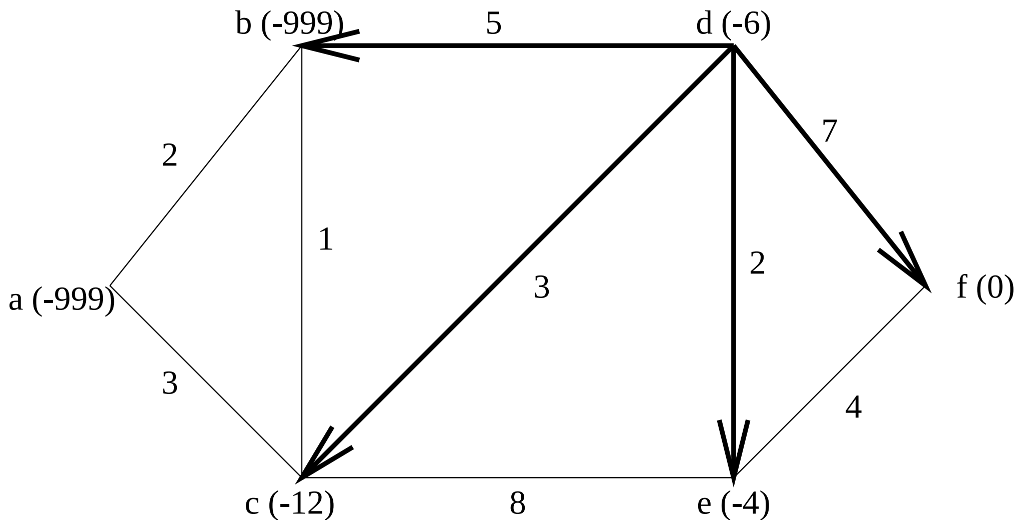


Value Iteration (2)

Choose State **c** again:

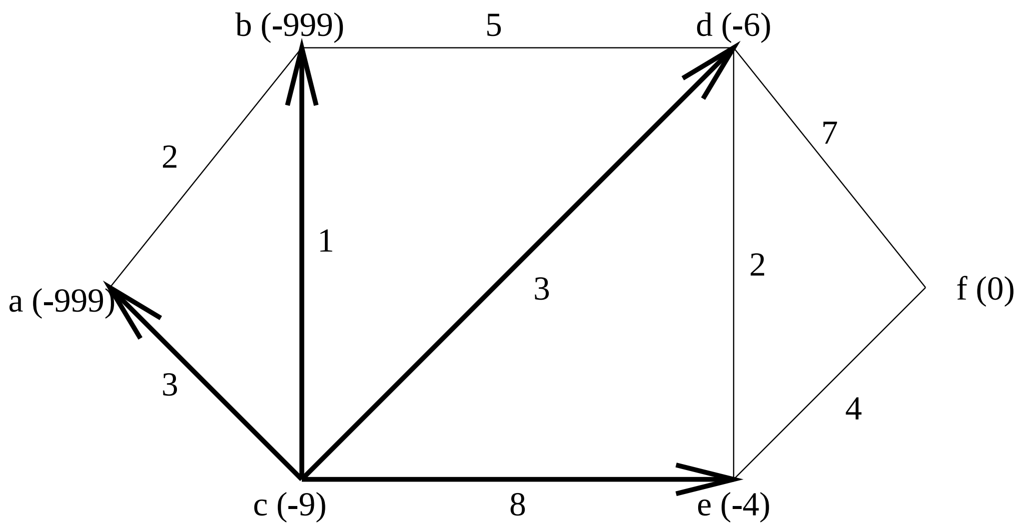


Choose State **d**:

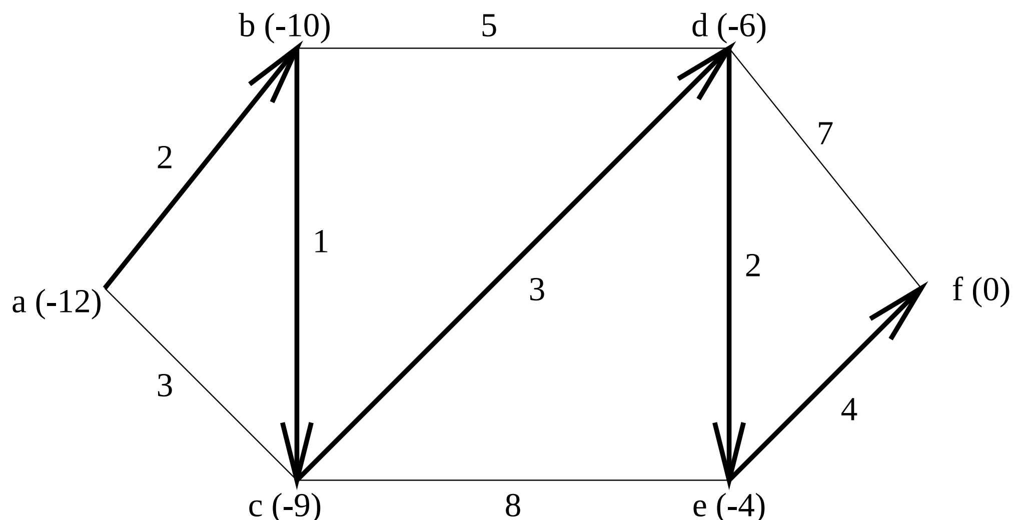


Value Iteration (3)

Choose State **c** a third time:



Final Value Function (and policy):

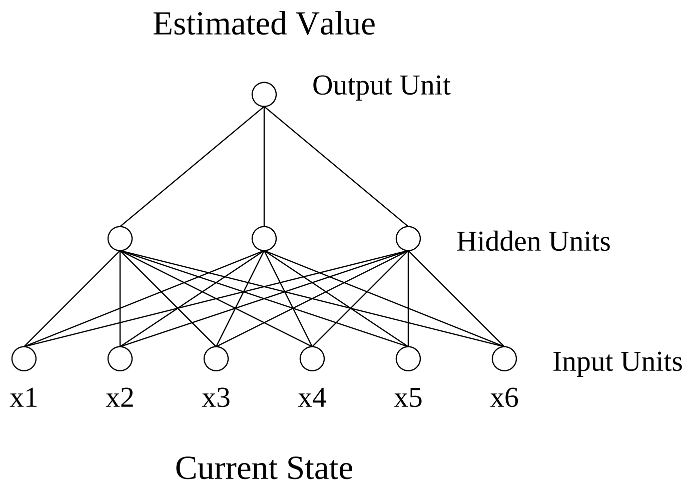


Value Iteration (4)

- **Value Iteration propagates information backwards from the goal**
- **The "backups" can be done in any order**
- **If done infinitely often in every state, the value function will converge to the optimal value function**
- **Can be extended to actions that have random effects.**
- **Problem: Requires storing a value of f for every state s.**

Temporal Difference Learning

Instead of storing the values of f in a large table, why not use a neural network?



- **Train neural network to reduce the** *temporal difference***:**

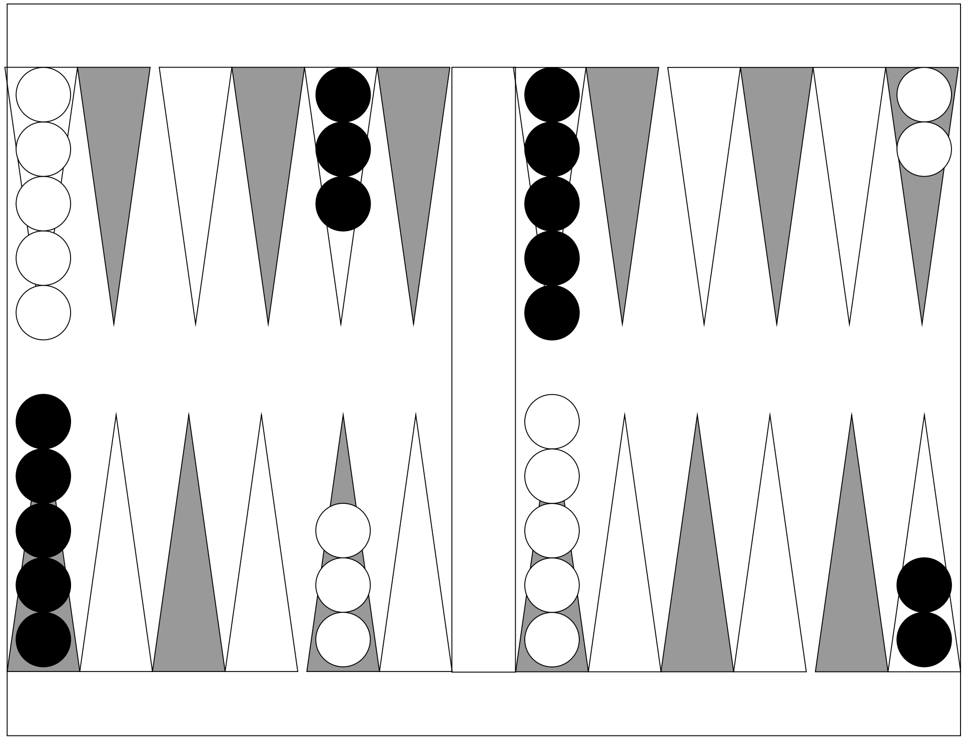
$$\hat{f}(s) - [\max_a \ r(a(s)) + \hat{f}(a(s))]$$

- **Use the backpropagation algorithm (gradient descent)**

$$J(s, \mathbf{W}) = \left(\hat{f}(s, \mathbf{W}) - [\max_a \ r(a(s)) + \hat{f}(a(s), \mathbf{W})]\right)^2$$

Example: Learning to Play Backgammon

- **State:** Board position + results of dice roll.
- **Actions:** Choose which pieces to move and by how much.
- **Reward:** Zero for all states except terminal state: $+1$ for win, -1 for loss.



Tesauro's TD-Gammon

Gerry Tesauro (IBM Research) trained a network using $TD(\lambda)$ to play world-class backgammon.

Program	Training Games	Opponents	Results
TDG 1.0	300,000	Robertie, Davis, Magriel	-13 pts/51 games (-0.25 ppg)
TDG 2.0	800,000	Goulding, Woolsey Snellings, Russell, Sylvester	-7 pts/38 games (-0.18 ppg)
TDG 2.1	1,500,000	Robertie	-1 pts/40 games (-0.02 ppg)

Formulating Scheduling as Reinforcement Learning

Goals: (a) reward short schedules (b) penalize long searches (c) normalize values.

- **Reward –0.001 each action**.
- – **(schedule length)**.

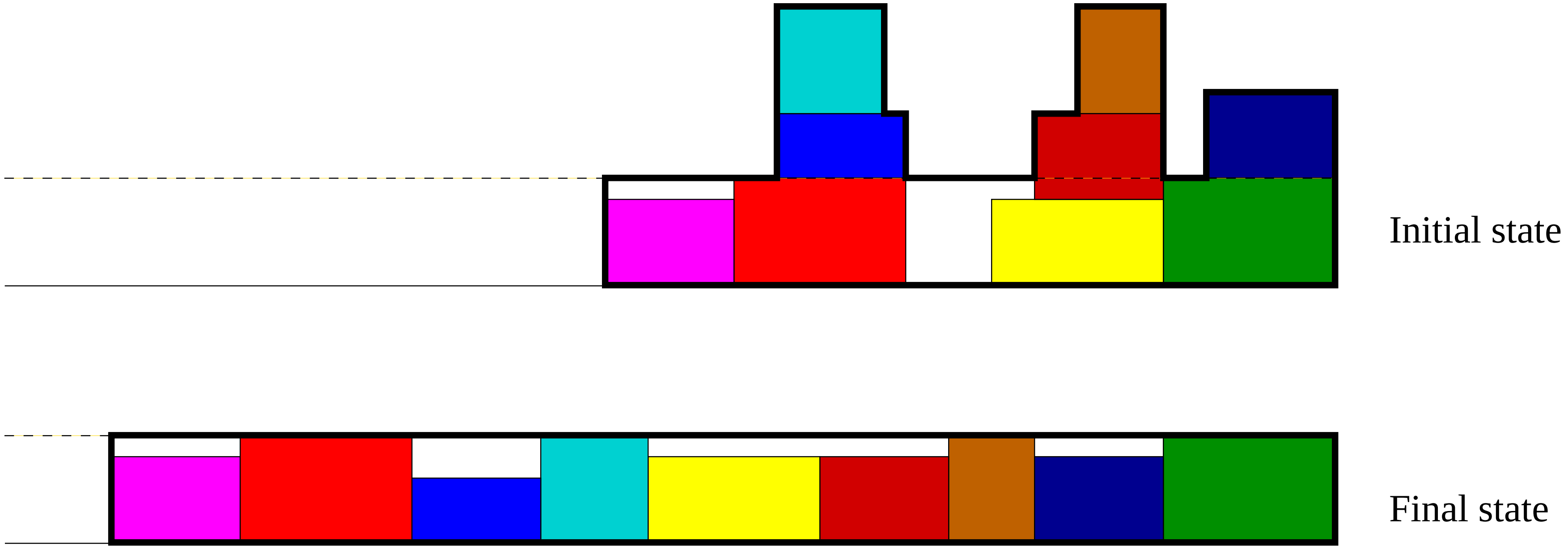
  Different problems get very different final rewards.

  Hard for network to predict.
- – **(schedule length/initial schedule length)**

  A hard problem may have a short initial schedule.
- – **(final resource reservation/initial resource reservations)**

  RDF: *Resource Dilation Factor*

RDF: Resource Dilation Factor



Input Features and the Value Function Network

- **Mean and standard deviation of the free pool capacity for bottleneck pools**
- **Mean and standard deviation of slacks**
- **Modified RDF**
- **Over-allocation index**
- **Windows in violation**
- **Percentage of windows in violation that can be resolved by pool reassignment**
- **Percentage of time units in violation**
- **First violated window index (normalized)**
- **Critical path schedule resource utilization**

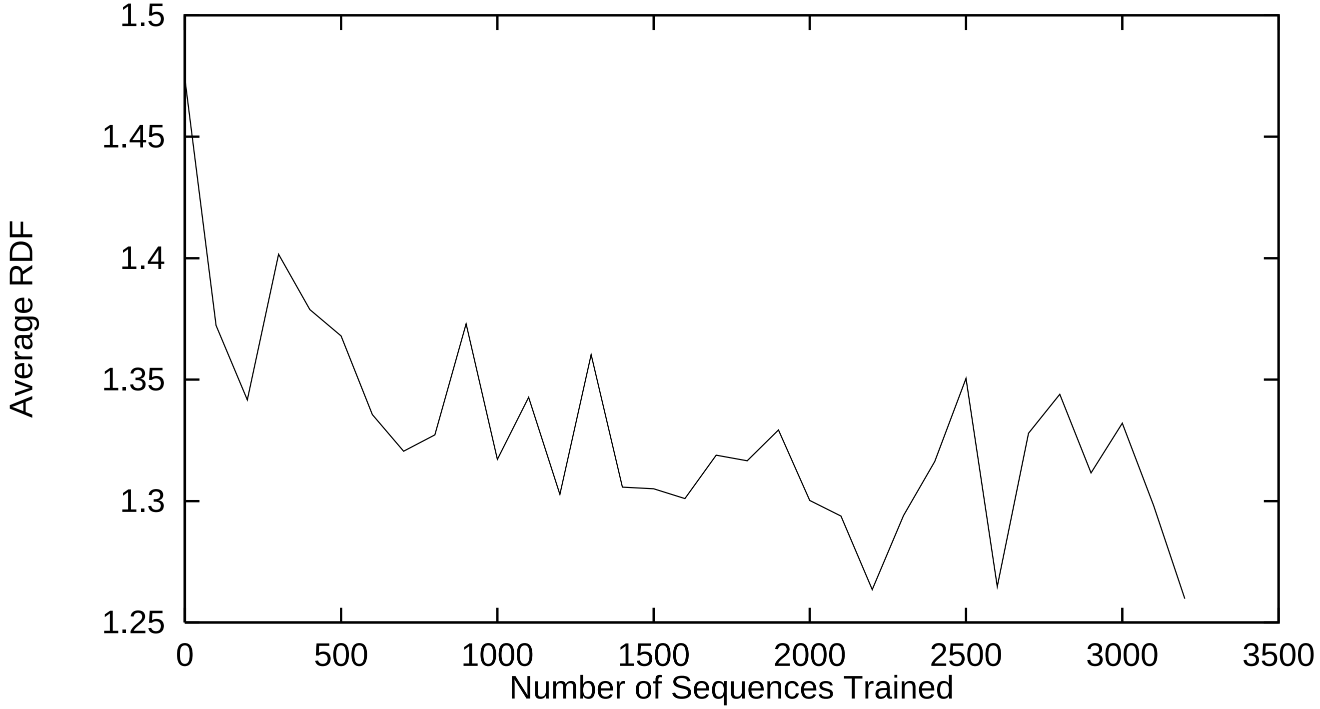
These features were input to a sigmoid neural network with 40 hidden units and 8 outputs (encoding the value using Gaussian overlapping fields).

Formulating Scheduling as Reinforcement Learning

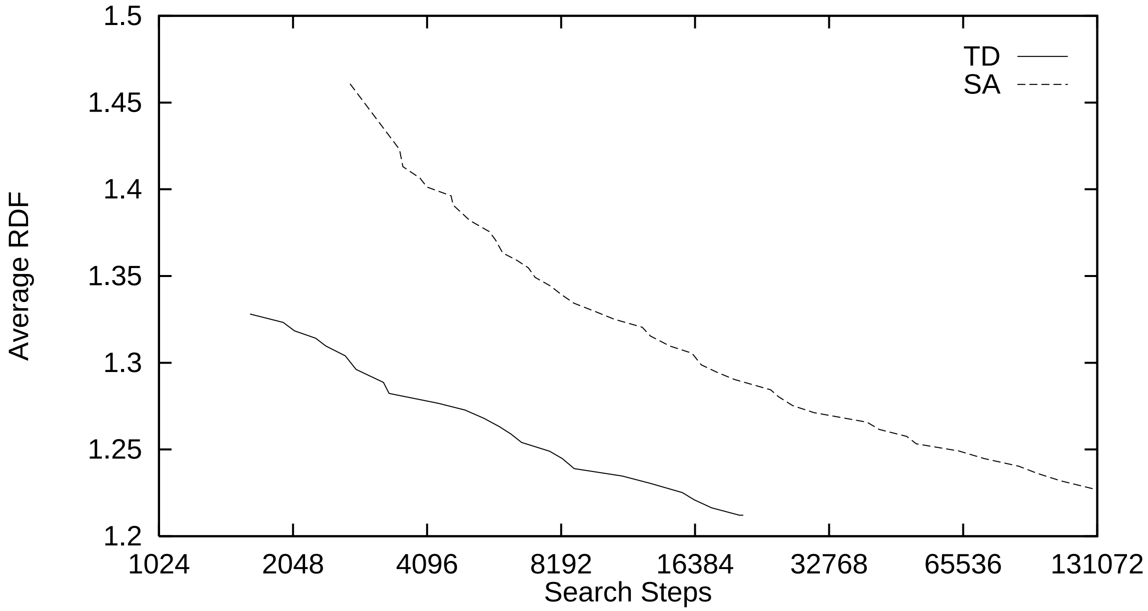
- **Random initial weights: Initial policy is random**
- **Actions selected by one-step greedy lookahead search**
- **Exploration: Random action chosen with probability** β
  (initially 1, gradually decreased by steps of 0.001 to 0.05).
- **Experience Replay.**
  Remember the sequence of actions and states (trajectory). When final state is reached, do updates in reverse order along the trajectory.
- **Best Trajectory Replay.**
  Remember the best trajectory for each problem. Every 4 trials, replay the best trajectory (doing updates in reverse order).
- **Overtraining Big Errors.**
  When the network makes a large temporal difference error, do several gradient descent steps for that state.

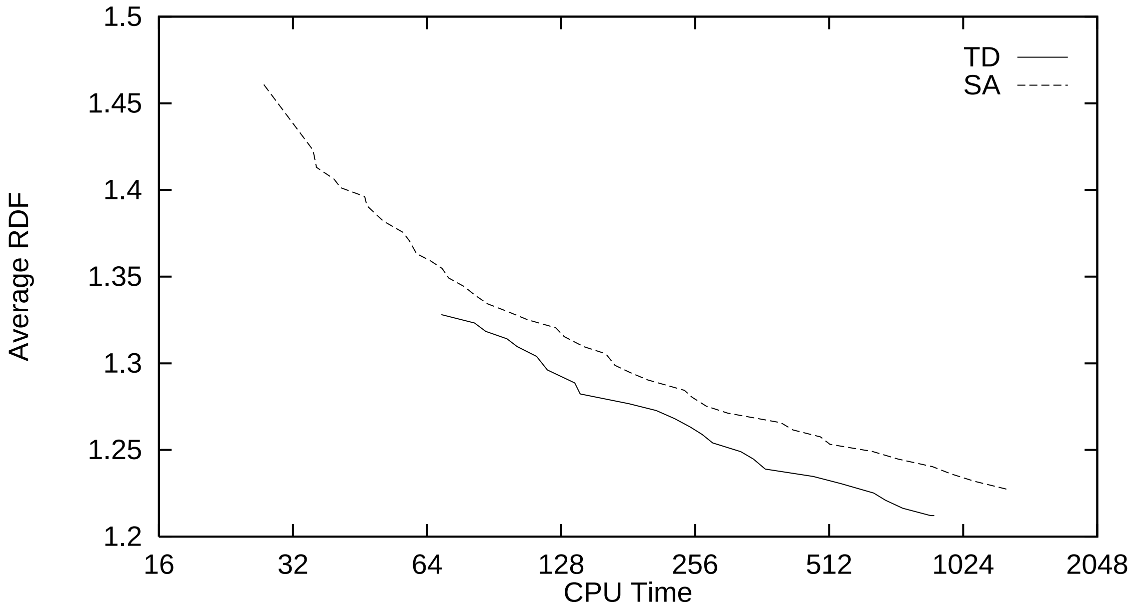
Training is Successful

Generated 100 problems: 30 training, 20 validation, 50 test.



Test Results





Why Does It Work?

- **Hypothesis 1: Value Iteration!**

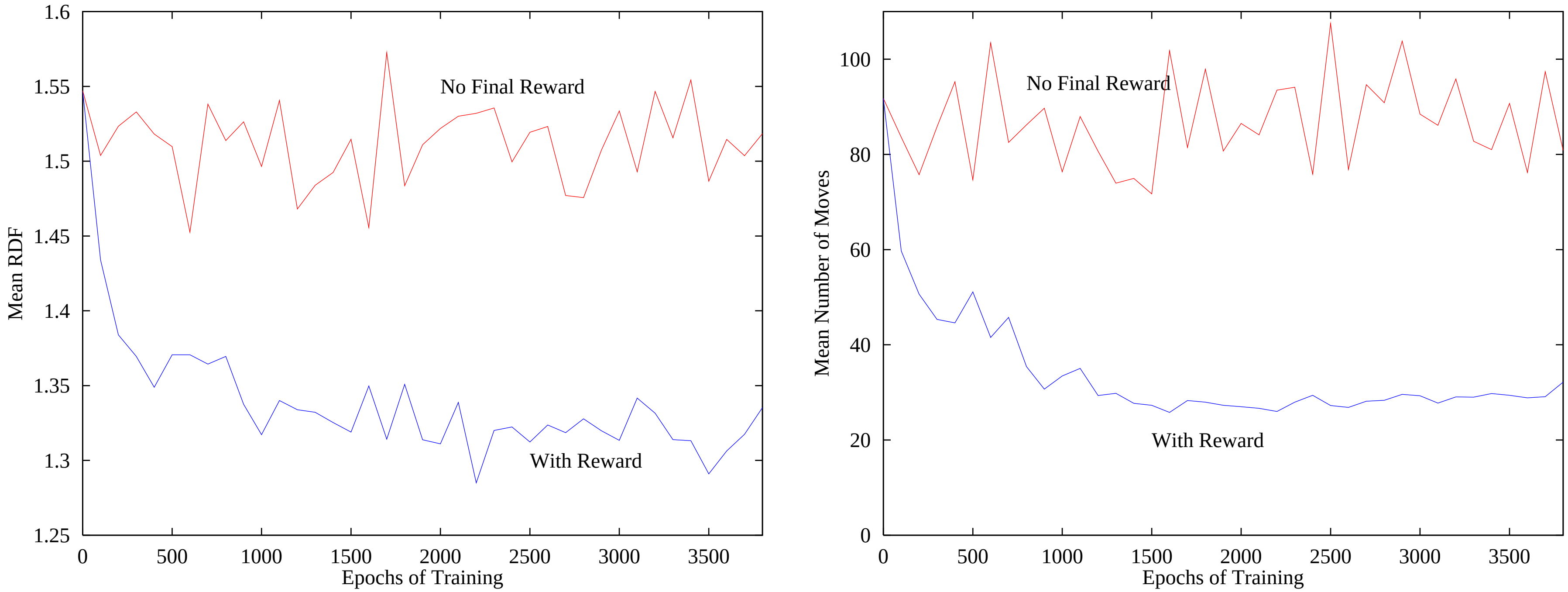
  Value iteration is successfully propagating the final reward backward toward the start state.

- **Hypothesis 2: Feature Engineering!**

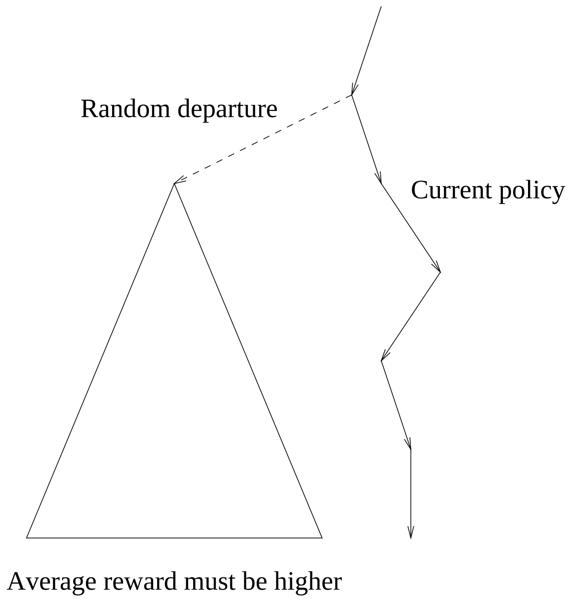
  The input features are good predictors by themselves. The network is just smoothing them.

Evidence For Value Iteration

- **Experiment: Train with no final reward**



When Will $TD(\lambda)$ Work?



- **A random walk must be able to reach a reward quickly.**

  Otherwise, no reward is ever received.

- **A (superior) random departure from the current policy must be detectable with high probability**

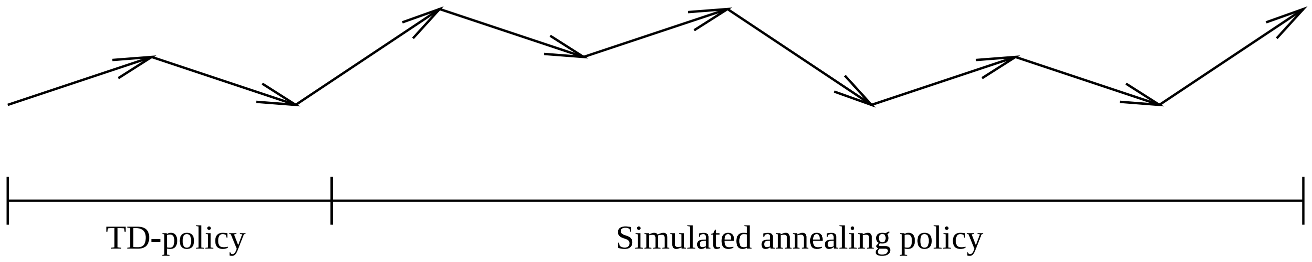
  Otherwise, the policy will not be changed.

Progressive Problem Spaces

- **Even random moves make progress.**
- **Early decisions are not undone by later ones.**

Both scheduling and backgammon are progressive problem spaces.

Experimental Confirmation



- **Prediction:** TD will give the same improvement no matter where it is inserted.