6.231 DYNAMIC PROGRAMMING

LECTURE 14

LECTURE OUTLINE

- Limited lookahead policies
- Performance bounds
- Computational aspects
- Problem approximation approach
- Vehicle routing example
- Heuristic cost-to-go approximation
- Computer chess

LIMITED LOOKAHEAD POLICIES

• *One-step lookahead (1SL) policy*: At each k and state x_k, use the control $\overline{\mu}_k(x_k)$ that

$$\min_{u_k \in U_k(x_k)} E\Big\{g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k, w_k)\big)\Big\},$$

where

– $\tilde{J}_N = g_N$.

– $\tilde{J}_{k+1}$: approximation to true cost-to-go J_{k+1}

• *Two-step lookahead policy*: At each k and x_k, use the control $\tilde{\mu}_k(x_k)$ attaining the minimum above, where the function $\tilde{J}_{k+1}$ is obtained using a 1SL approximation (solve a 2-step DP problem).

• If $\tilde{J}_{k+1}$ is readily available and the minimization above is not too hard, the 1SL policy is implementable on-line.

• Sometimes one also replaces $U_k(x_k)$ above with a subset of "most promising controls" $\overline{U}_k(x_k)$.

• As the length of lookahead increases, the required computation quickly explodes.

PERFORMANCE BOUNDS

- Let $\overline{J}_k(x_k)$ be the cost-to-go from (x_k, k) of the 1SL policy, based on functions $\tilde{J}_k$.
- Assume that for all (x_k, k), we have

$$\hat{J}_k(x_k) \leq \tilde{J}_k(x_k), \qquad (*)$$

where $\hat{J}_N = g_N$ and for all k,

$$\hat{J}_k(x_k) = \min_{u_k \in U_k(x_k)} E\Big\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k, w_k)\big) \Big\},$$

[so $\hat{J}_k(x_k)$ is computed along with $\overline{\mu}_k(x_k)$]. Then

$$\overline{J}_k(x_k) \leq \hat{J}_k(x_k), \qquad \text{for all } (x_k, k).$$

- Important application: When $\tilde{J}_k$ is the cost-to-go of some heuristic policy (then the 1SL policy is called the *rollout* policy).
- The bound can be extended to the case where there is a δ_k in the RHS of (*). Then

$$\overline{J}_k(x_k) \leq \tilde{J}_k(x_k) + \delta_k + \cdots + \delta_{N-1}$$

COMPUTATIONAL ASPECTS

- Sometimes nonlinear programming can be used to calculate the 1SL or the multistep version [particularly when $U_k(x_k)$ is not a discrete set]. Connection with the methodology of stochastic programming.

- The choice of the approximating functions $\tilde{J}_k$ is critical, and is calculated with a variety of methods.

- Some approaches:

  (a) *Problem Approximation*: Approximate the optimal cost-to-go with some cost derived from a related but simpler problem

  (b) *Heuristic Cost-to-Go Approximation*: Approximate the optimal cost-to-go with a function of a suitable parametric form, whose parameters are tuned by some heuristic or systematic scheme (Neuro-Dynamic Programming)

  (c) *Rollout Approach*: Approximate the optimal cost-to-go with the cost of some suboptimal policy, which is calculated either analytically or by simulation

PROBLEM APPROXIMATION

- Many (problem-dependent) possibilities
  - Replace uncertain quantities by nominal values, or simplify the calculation of expected values by limited simulation
  - Simplify difficult constraints or dynamics

- Example of *enforced decomposition*: Route m vehicles that move over a graph. Each node has a "value." The first vehicle that passes through the node collects its value. Max the total collected value, subject to initial and final time constraints (plus time windows and other constraints).

- Usually the 1-vehicle version of the problem is much simpler. This motivates an approximation obtained by solving single vehicle problems.

- 1SL scheme: At time k and state x_k (position of vehicles and "collected value nodes"), consider all possible kth moves by the vehicles, and at the resulting states we approximate the optimal value-to-go with the value collected by optimizing the vehicle routes one-at-a-time

HEURISTIC COST-TO-GO APPROXIMATION

- Use a cost-to-go approximation from a parametric class $\tilde{J}(x, r)$ where x is the current state and $r = (r_1, \ldots, r_m)$ is a vector of "tunable" scalars (weights).

- By adjusting the weights, one can change the "shape" of the approximation $\tilde{J}$ so that it is reasonably close to the true optimal cost-to-go function.

- Two key issues:
  - The choice of parametric class $\tilde{J}(x, r)$ (the approximation architecture).
  - Method for tuning the weights ("training" the architecture).

- Successful application strongly depends on how these issues are handled, and on insight about the problem.

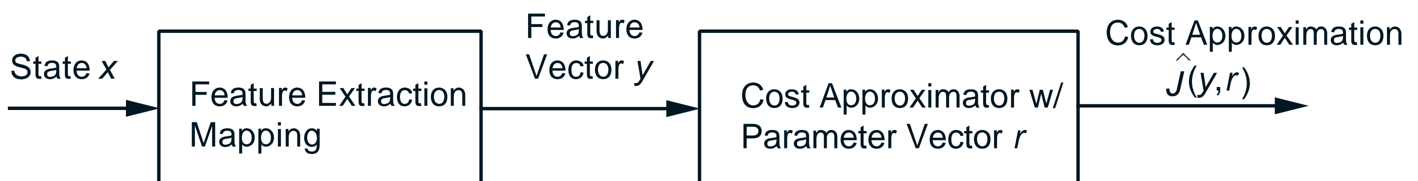
- Sometimes a simulator is used, particularly when there is no mathematical model of the system.

APPROXIMATION ARCHITECTURES

- Divided in linear and nonlinear [i.e., linear or nonlinear dependence of $\tilde{J}(x,r)$ on r].

- Linear architectures are easier to train, but nonlinear ones (e.g., neural networks) are richer.

- Architectures based on feature extraction



- Ideally, the features will encode much of the nonlinearity that is inherent in the cost-to-go approximated, and the approximation may be quite accurate without a complicated architecture.

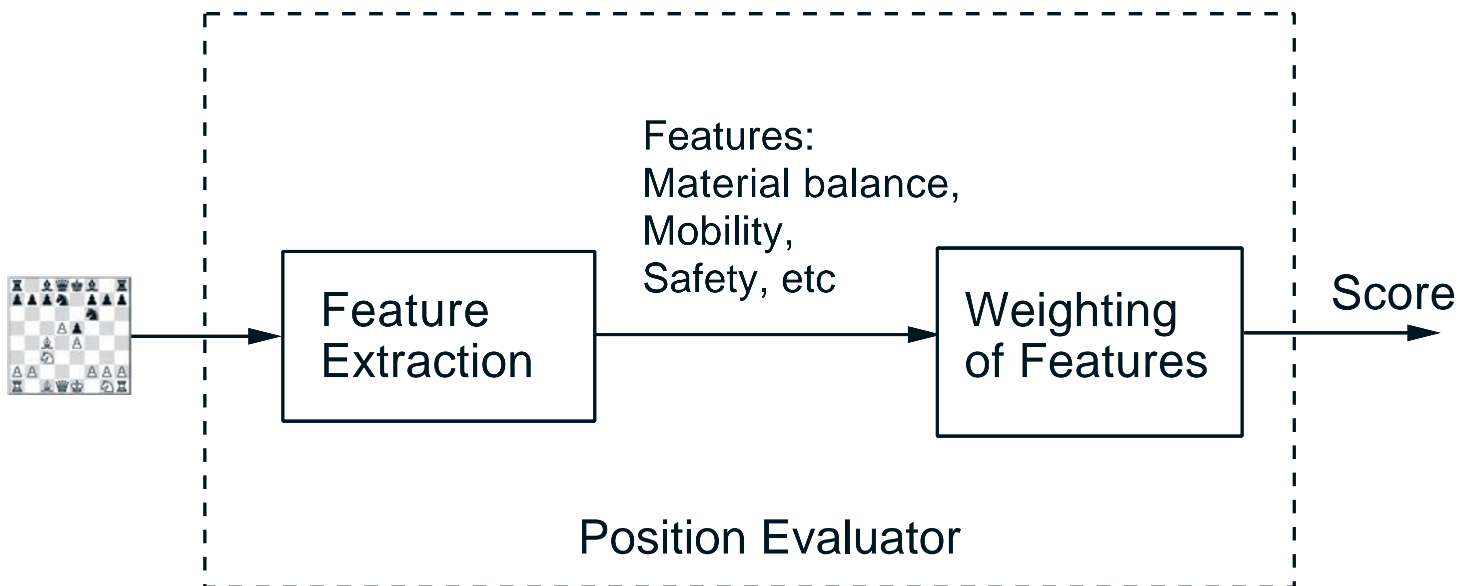
- Sometimes the state space is partitioned, and "local" features are introduced for each subset of the partition (they are 0 outside the subset).

- With a well-chosen feature vector $y(x)$, we can use a linear architecture

$$\tilde{J}(x,r) = \hat{J}\big(y(x),r\big) = \sum_i r_i y_i(x)$$

COMPUTER CHESS I

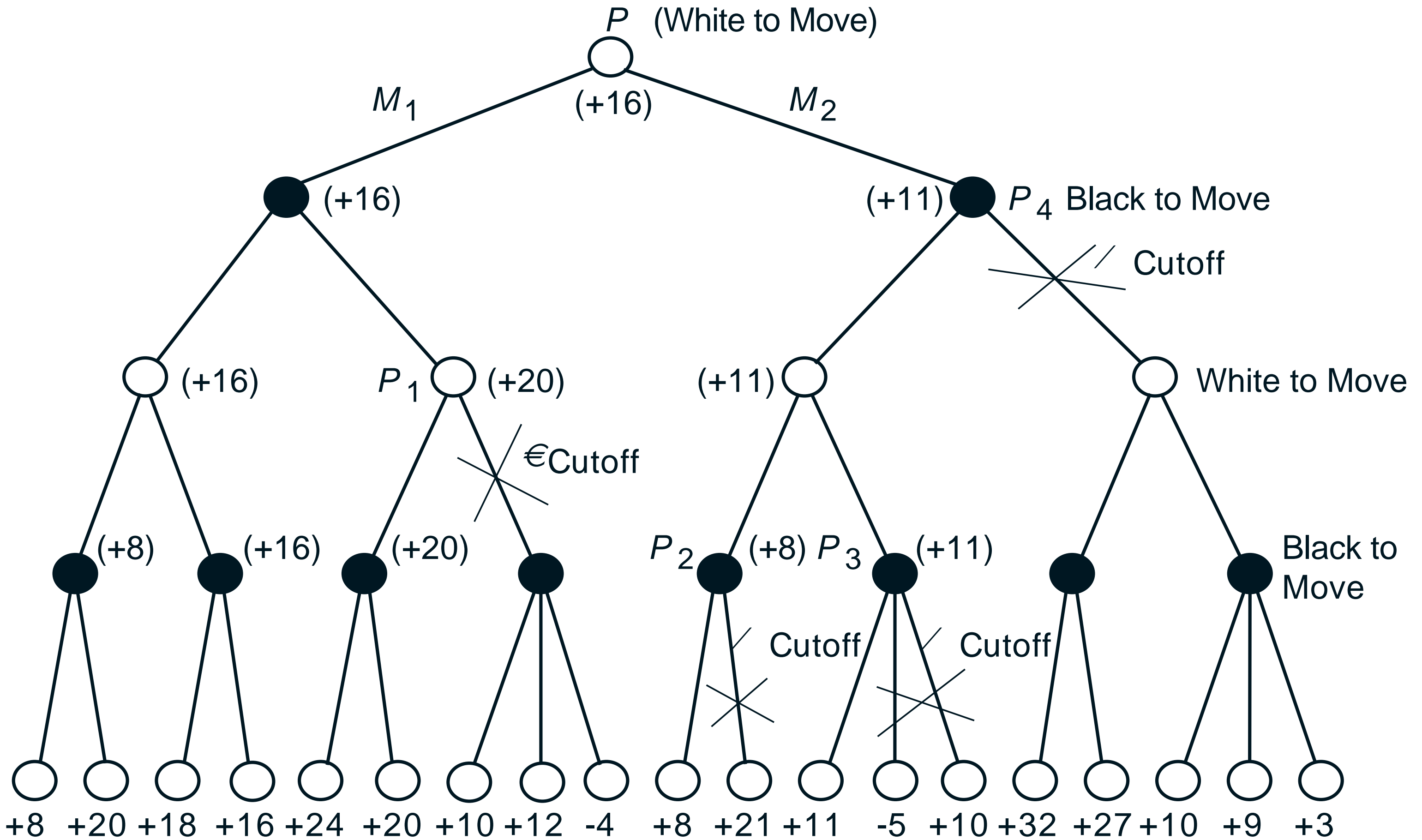
- Programs use a feature-based position evaluator that assigns a score to each move/position



- Most often the weighting of features is linear but multistep lookahead is involved.
- Most often the training is done by trial and error.
- Additional features:
  - Depth first search
  - Variable depth search when dynamic positions are involved
  - Alpha-beta pruning

COMPUTER CHESS II

- Multistep lookahead tree



- Alpha-beta pruning: As the move scores are evaluated by depth-first search, branches whose consideration (based on the calculations so far) cannot possibly change the optimal move are neglected