Game Playing: Adversarial Search

Minimax - Nim

- Assuming MIN plays first, complete the MIN/MAX tree
- Assume that a utility function of
  - 0 = a win for MIN
  - 1 = a win for MAX

Minimax Rule

- Goal of game tree search: to determine **one move** for Max player that **maximizes** the **guaranteed payoff** for a given game tree for MAX

  Regardless of the moves the MIN will take

- The value of each node (Max and MIN) is determined by (back up from) the values of its children

- MAX plays the worst case scenario:

  Always assume MIN to take moves to maximize his pay-off (i.e., to minimize the pay-off of MAX)

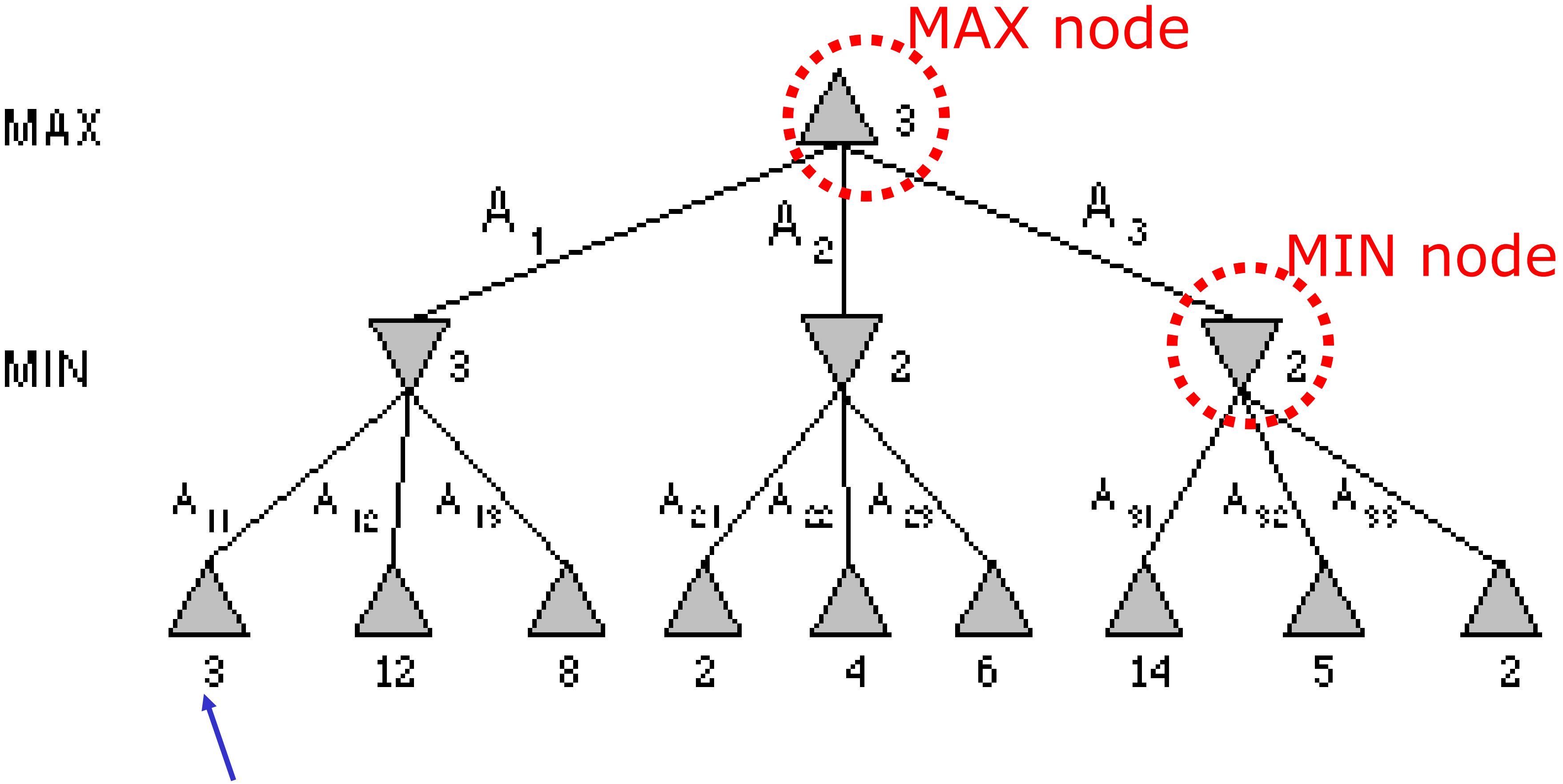
- For a MAX node, the backed up value is the **maximum** of the values associated with its children

- For a MIN node, the backed up value is the **minimum** of the values associated with its children

Minimax to a Fixed Ply-Depth

- Usually not possible to expand a game to end-game status
  - have to choose a ply-depth that is achievable with reasonable time and resources
  - absolute ‘win-lose’ values become heuristic scores
  - heuristics are devised according to knowledge of the game

Minimax Tree



MAX node

MIN node

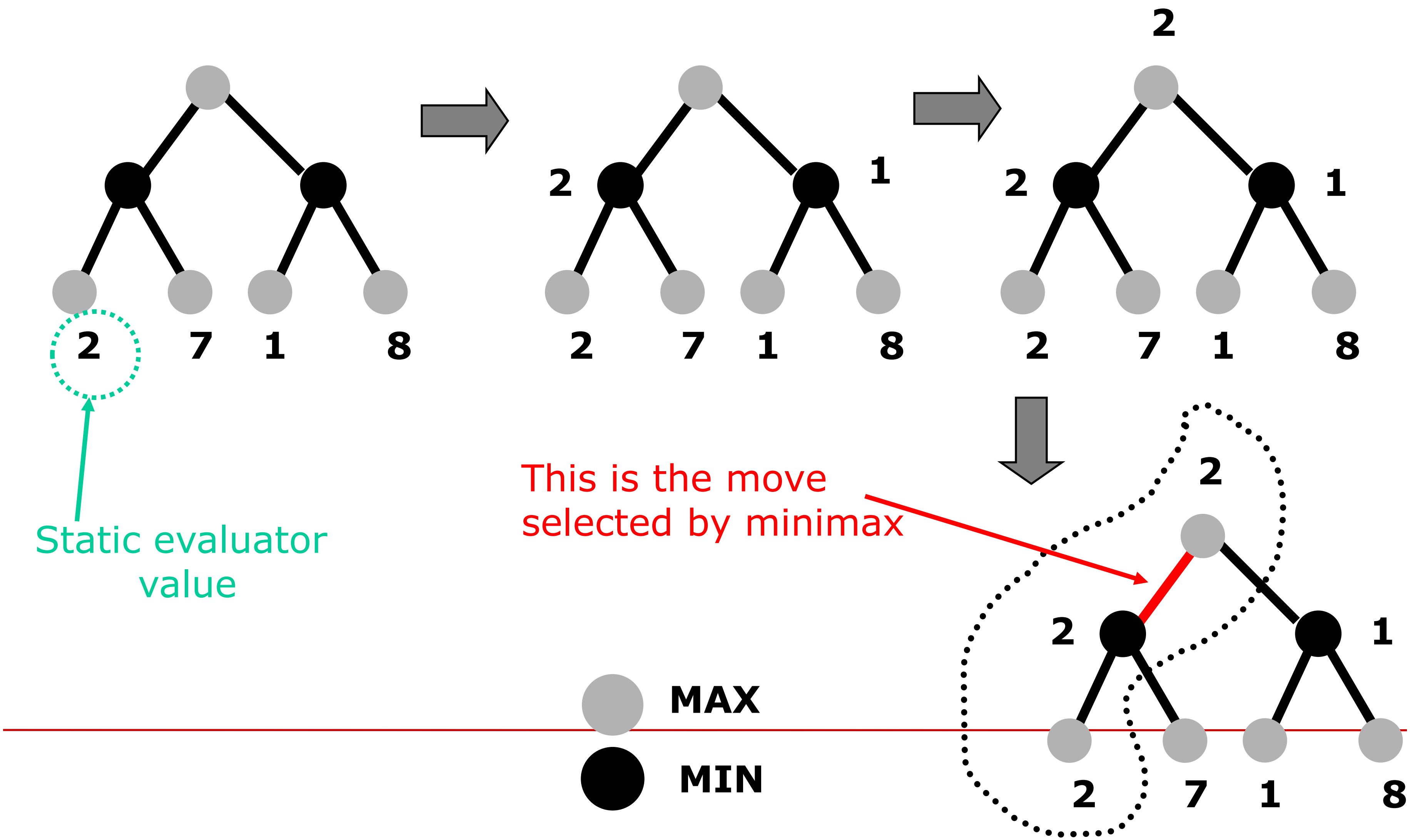
f value

A$_1$ is selected as the next move

Minimax procedure

- Create start node as a MAX node with current game configuration
- Expand nodes down to some depth (i.e., ply) of lookahead in the game.
- Apply the evaluation function at each of the leaf nodes
- Obtain the “back up" values for each of the non-leaf nodes from its children by Minimax rule until a value is computed for the root node.
- Pick the operator associated with the child node whose backed up value determined the value at the root as the move for MAX

Minimax Search



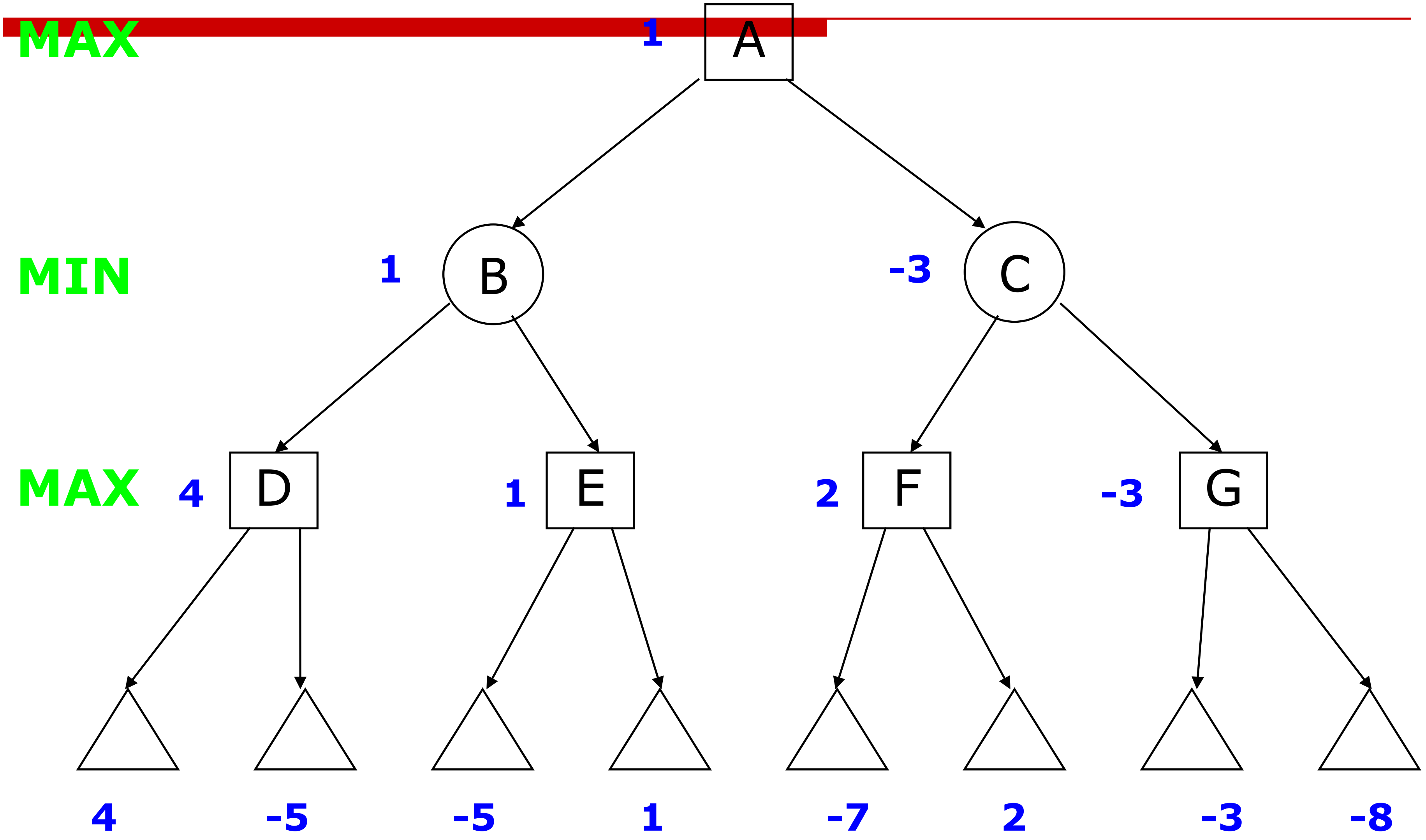
Static evaluator value

This is the move selected by minimax

MAX

MIN

Minimax - Example



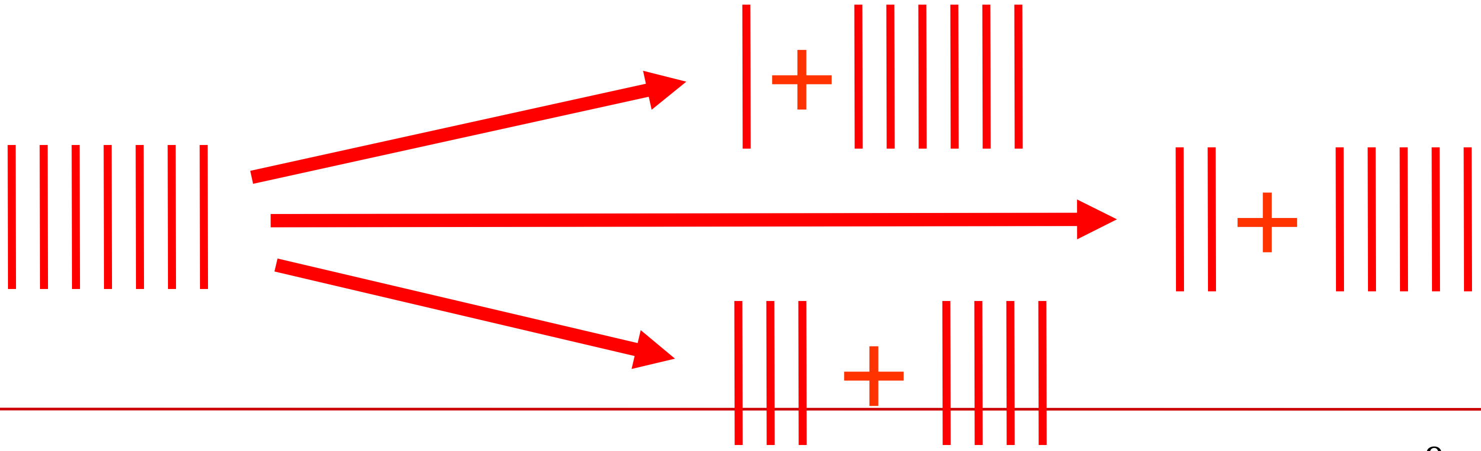
△ = terminal position

□ = agent

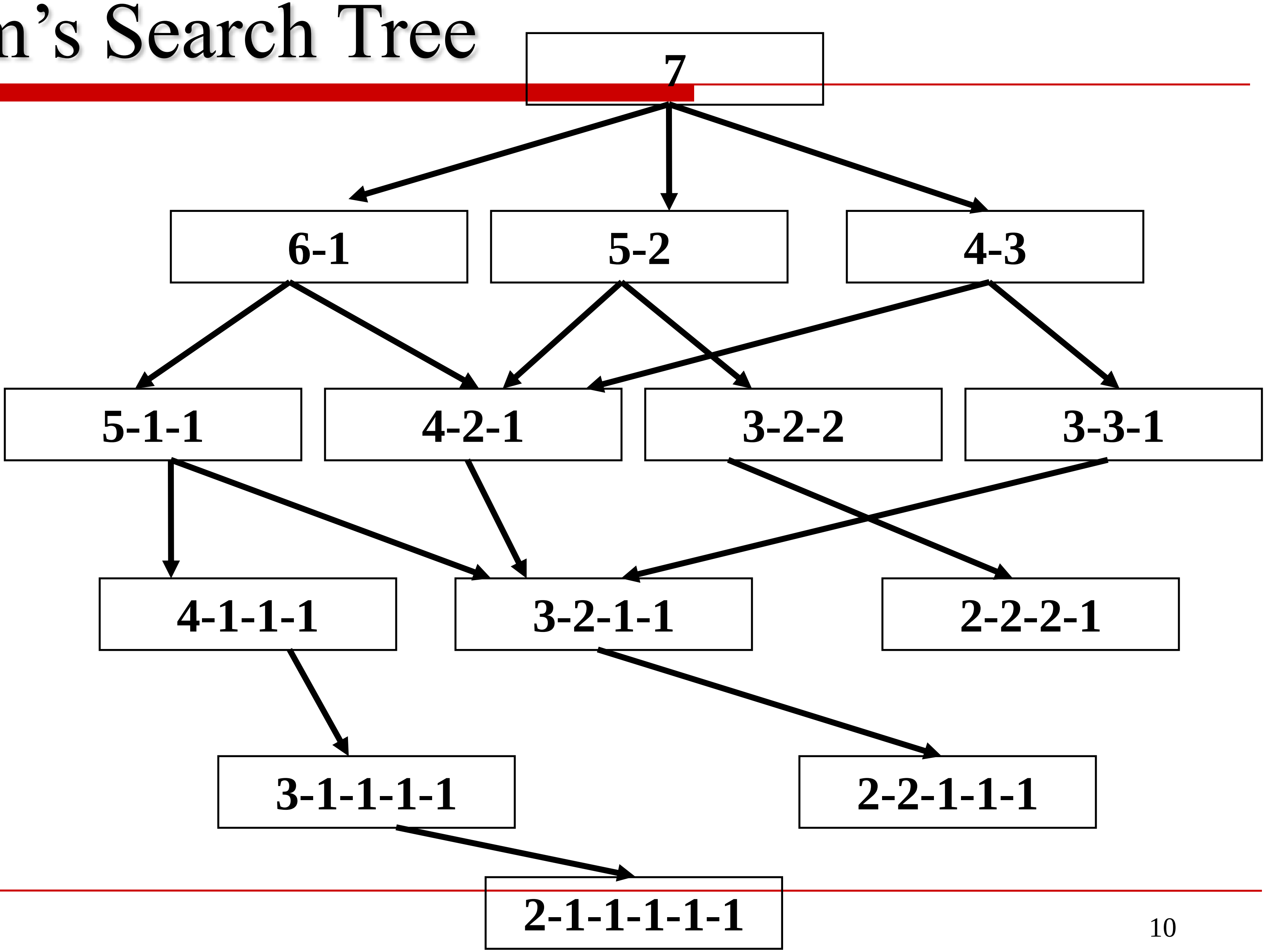
○ = opponent

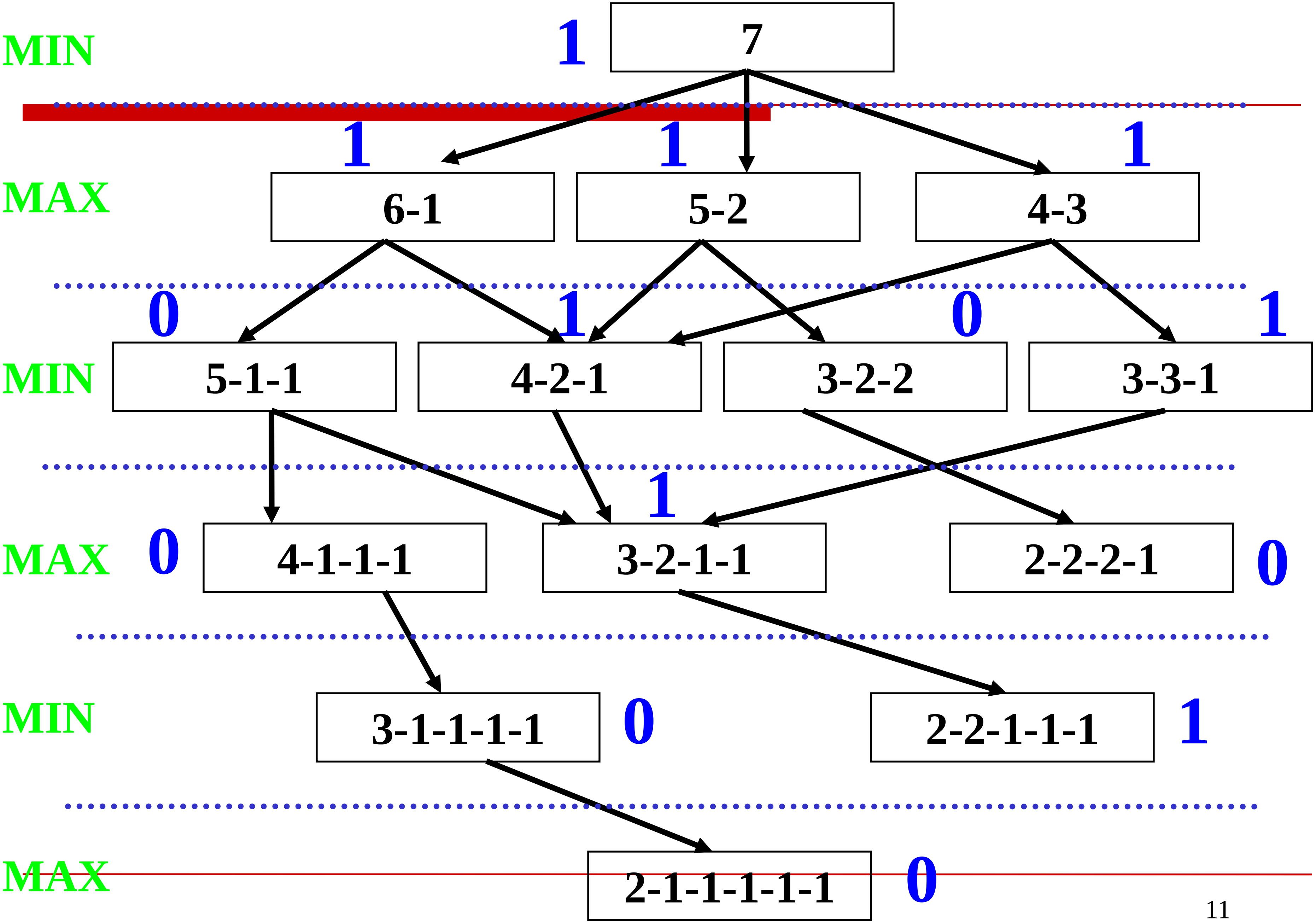
Minimax - Nim

- Nim
- Start with a pile of tokens
- At each move the player must divide the tokens into two non-empty, non-equal piles



Nim’s Search Tree





MIN 1 7

MAX 1 6-1 1 5-2 1 4-3

MIN 0 5-1-1 1 4-2-1 0 3-2-2 1 3-3-1

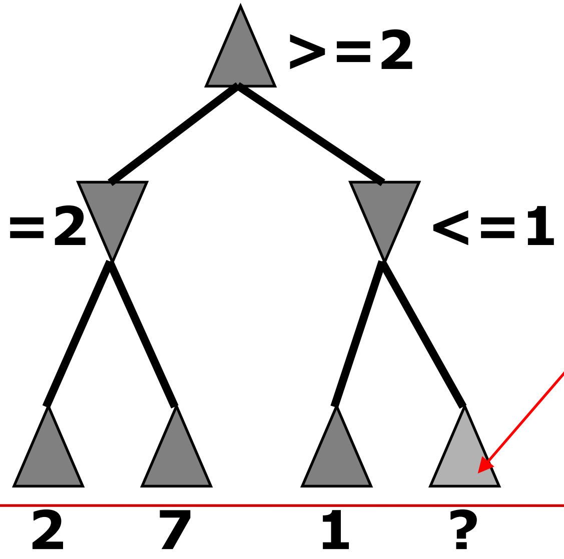
MAX 0 4-1-1-1 1 3-2-1-1 2-2-2-1 0

MIN 3-1-1-1-1 0 2-2-1-1-1 1

MAX 2-1-1-1-1-1 0

Alpha-beta pruning

- We can improve on the performance of the minimax algorithm through **alpha-beta** pruning.
- Basic idea: *"If you have an idea that is surely bad, don't take the time to see how truly awful it is."* -- Pat Winston



- We don't need to compute the value at this node.
- No matter what it is it can't effect the value of the root node.

Alpha-beta pruning

- **Alpha cutoff**: Given a Max node n, cutoff the search below n (i.e., don't generate or examine any more of n's children) if alpha(n) >= beta(n) (alpha increases and passes beta from below)

- **Beta cutoff.**: Given a Min node n, cutoff the search below n (i.e., don't generate or examine any more of n's children) if beta(n) <= alpha(n) (beta decreases and passes alpha from above)

- Carry alpha and beta values down during search

Pruning occurs whenever alpha >= beta

Alpha-beta pruning

- Traverse the search tree in depth-first order
- At each **Max** node n, **alpha(n)** = maximum value found so far
  - Start with -infinity and only increase
  - Increases if a child of n returns a value greater than the current alpha
  - Serve as a tentative lower bound of the final pay-off
- At each **Min** node n, **beta(n)** = minimum value found so far
  - Start with infinity and only decrease
  - Decreases if a child of n returns a value less than the current beta
  - Serve as a tentative upper bound of the final pay-off

Alpha-beta algorithm

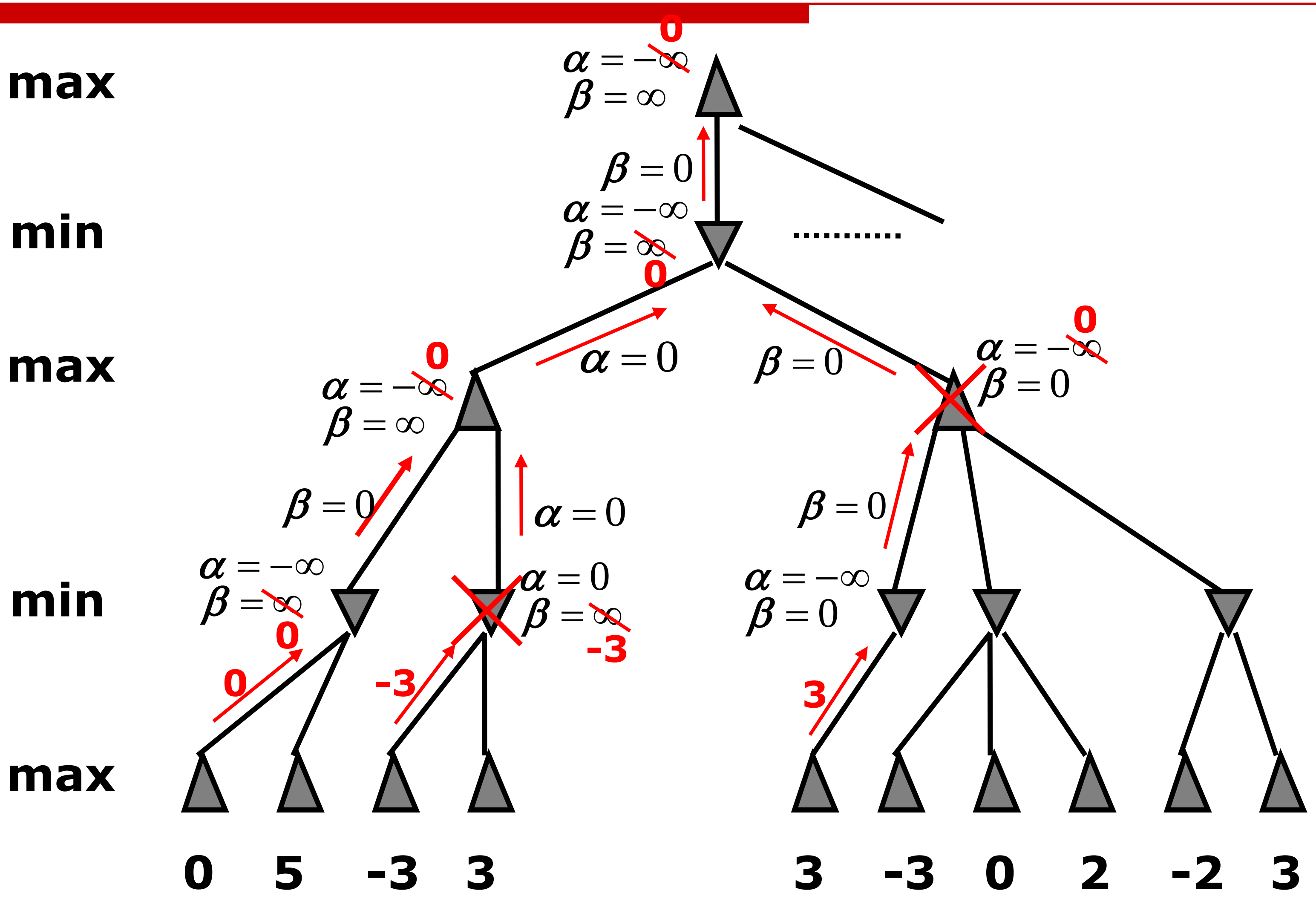
initiation
$\alpha := -\infty$
$\beta := \infty$

- Two functions recursively call each other

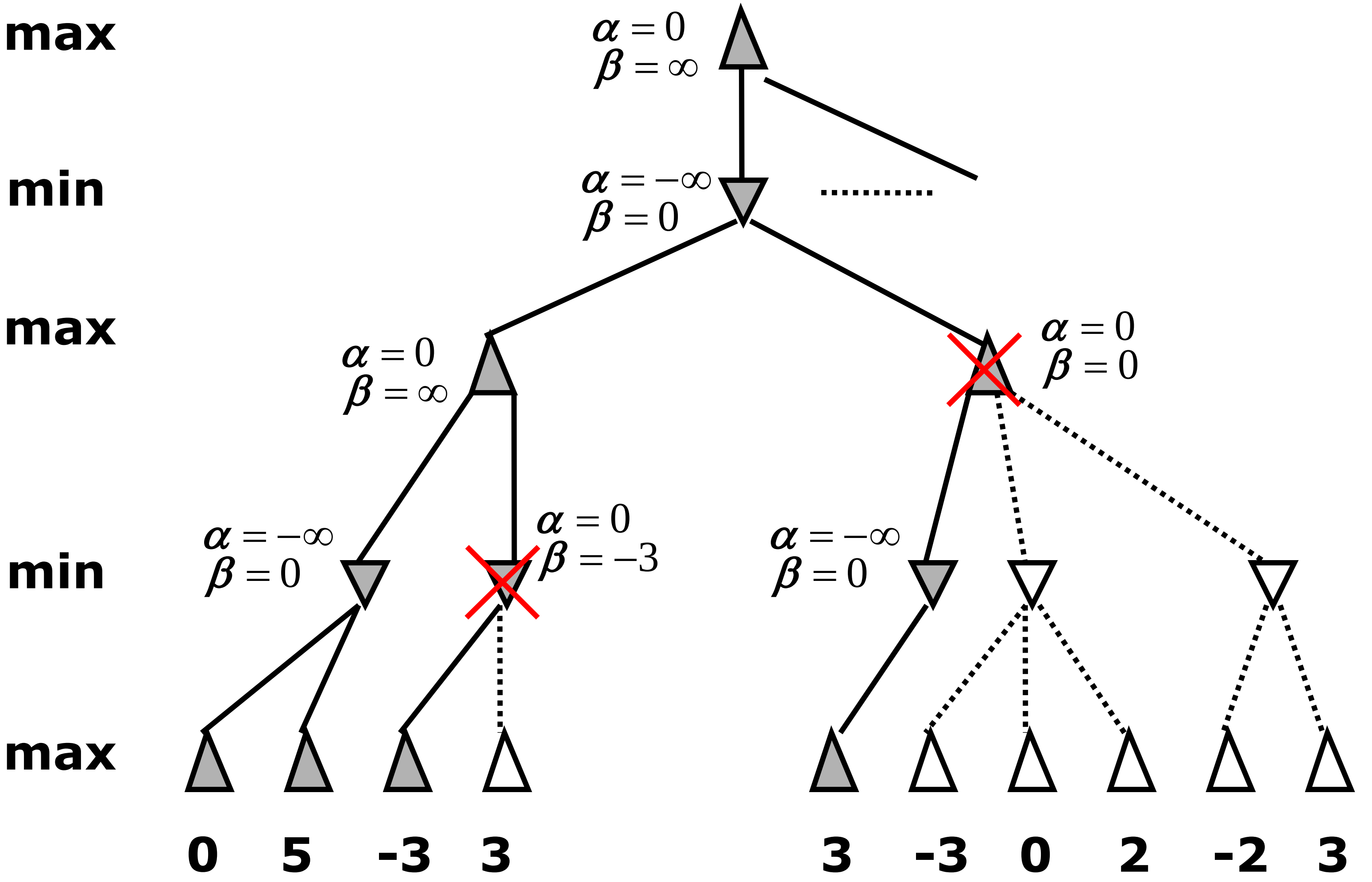
```
function MAX-value (n, alpha, beta)
   if n is a leaf node then return f(n);
   for each child n' of n do
         alpha :=max{alpha, MIN-value(n', alpha, beta)};
         if alpha >= beta then return beta   /* pruning */
   end{do}
   return alpha

function MIN-value (n, alpha, beta)
   if n is a leaf node then return f(n);
   for each child n' of n do
         beta :=min{beta, MAX-value(n', alpha, beta)};
         if beta <= alpha then return alpha   /* pruning */
   end{do}
   return beta
```

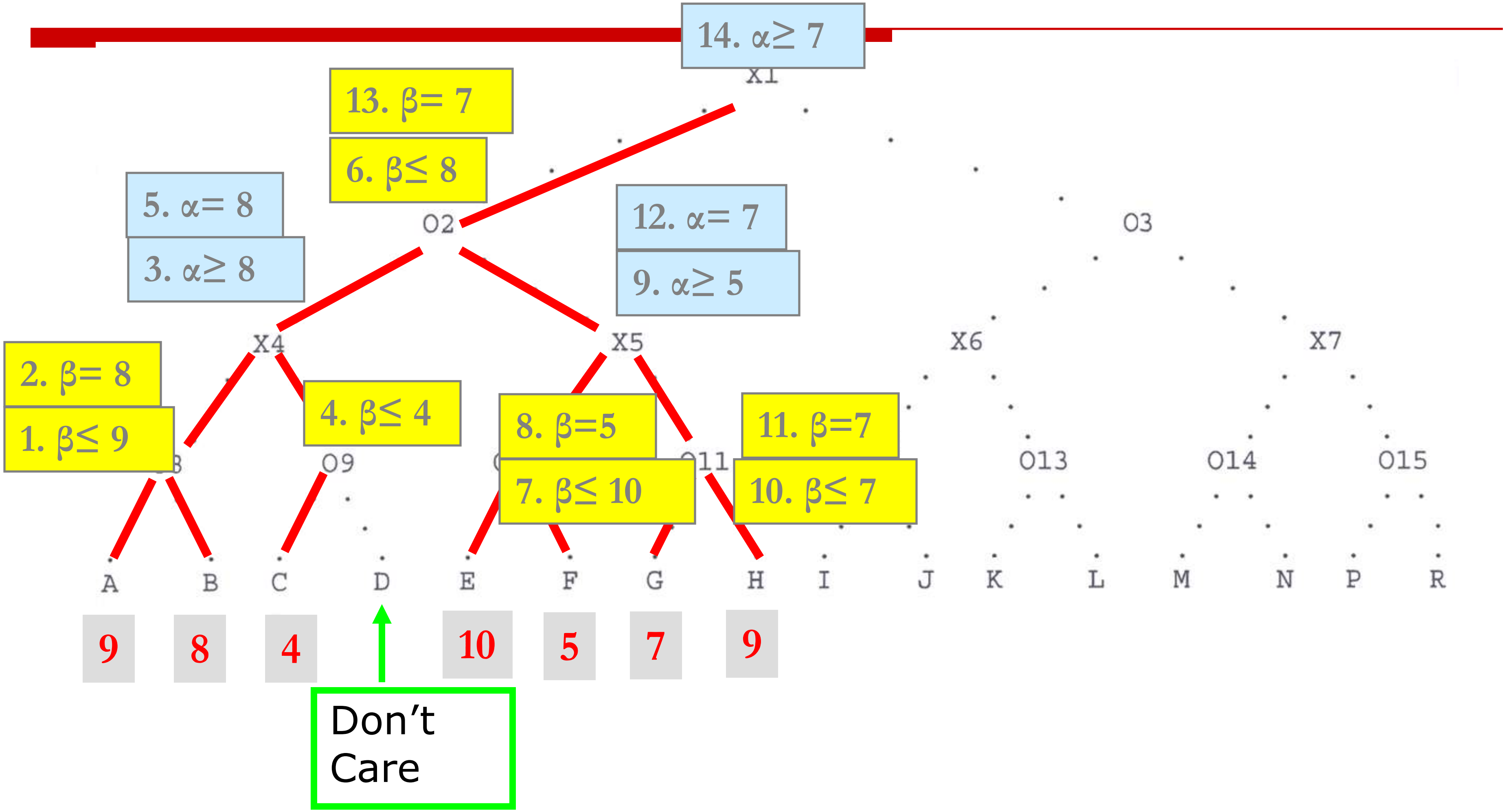
An example of Alpha-beta pruning



Final tree

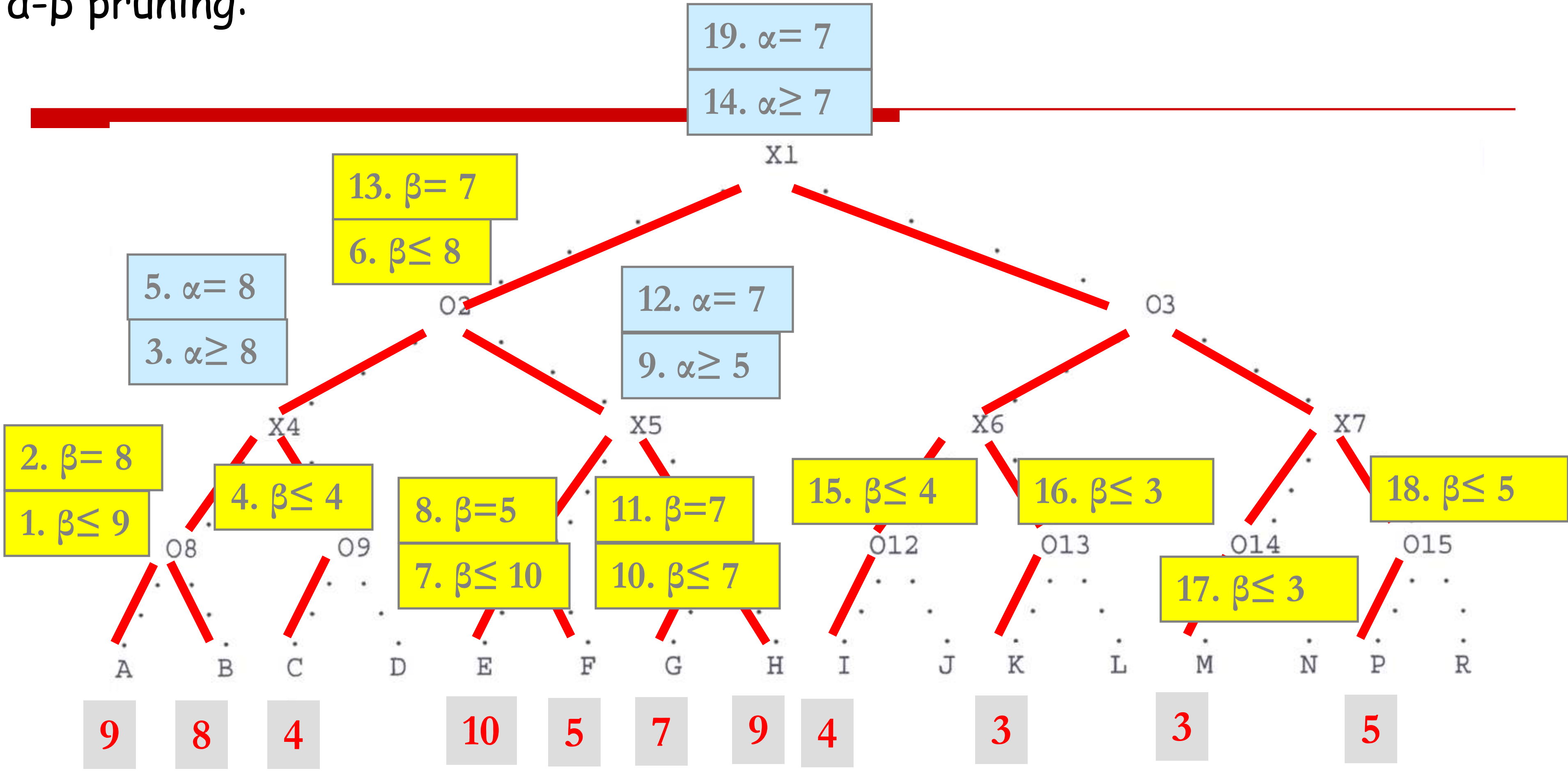

max
min
max
min
max
$\alpha = 0$
$\beta = \infty$
$\alpha = -\infty$
$\beta = 0$
....
$\alpha = 0$
$\beta = \infty$
$\alpha = 0$
$\beta = 0$
$\alpha = -\infty$
$\beta = 0$
$\alpha = 0$
$\beta = -3$
$\alpha = -\infty$
$\beta = 0$
0
5
-3
3
3
-3
0
2
-2
3

α-β pruning:

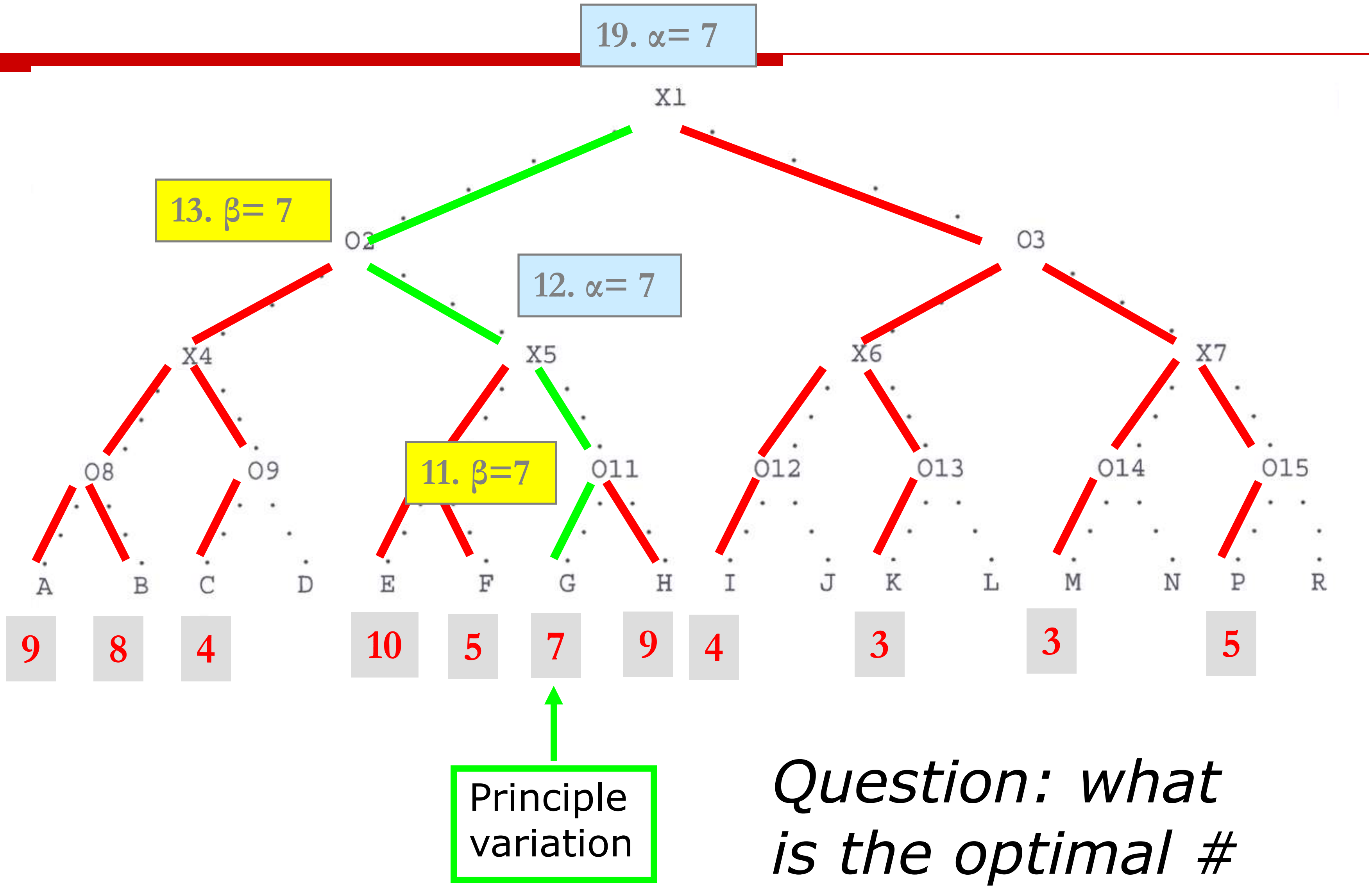


Depth-First Search to desired # of ply

α-β pruning:


19. α= 7
14. α≥ 7
X1
13. β= 7
6. β≤ 8
5. α= 8
3. α≥ 8
O2
12. α= 7
9. α≥ 5
O3
X4
X5
X6
X7
2. β= 8
1. β≤ 9
4. β≤ 4
8. β=5
7. β≤ 10
11. β=7
10. β≤ 7
15. β≤ 4
16. β≤ 3
18. β≤ 5
17. β≤ 3
O8
O9
O12
O13
O14
O15
A
B
C
D
E
F
G
H
I
J
K
L
M
N
P
R
9
8
4
10
5
7
9
4
3
3
5

α-β pruning:



Question: what is the optimal # of cutoffs?

Effectiveness of Alpha-beta pruning

- Alpha-Beta is guaranteed to compute the same value for the root node as computed by Minimax.
- **Worst case:** NO pruning, examining O(b^d) leaf nodes, where each node has b children and a d-ply search is performed
- **Best case:** examine only O(b^(d/2)) leaf nodes.
  - You can search twice as deep as Minimax! Or the branch factor is b^(1/2) rather than b.
- **Best case** is when each player's best move is the leftmost alternative, i.e. at MAX nodes the child with the largest value generated first, and at MIN nodes the child with the smallest value generated first.
- In Deep Blue, they found empirically that with Alpha-Beta pruning the average branching factor at each node was about 6 instead of about 35-40

Games of chance

- Game of chance is anything with a random factor; e.g., dice, cards, etc.

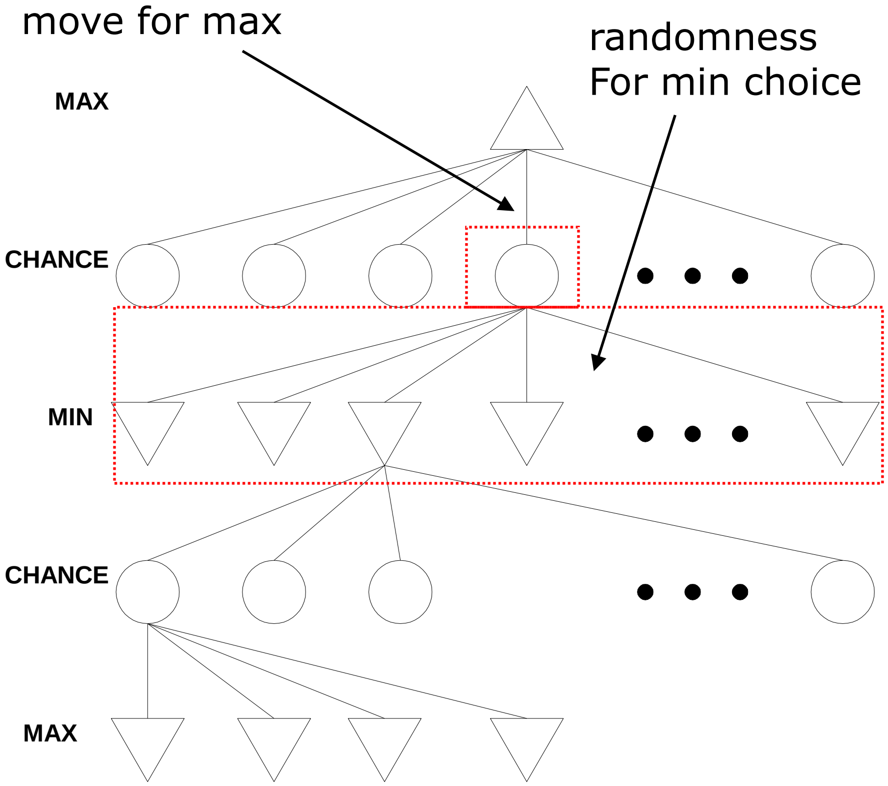
- Can extend the minimax method to handle games of chance.
  - This results in an **expectiminimax tree.**

- In an expectiminimax tree, levels of max and min nodes are **interleaved** with "chance" nodes.

- Rather than taking the max or min of the payoff values of their children, chance nodes take a weighted average (expected value) of their children.
  - Weights are the probability that the chance node is reached.
  - Each chance node gets an "expectiminimax value".

Game tree with chance nodes

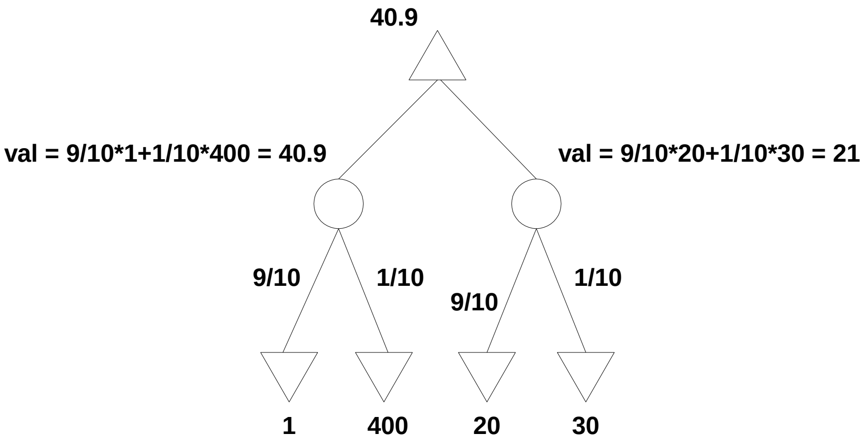
- Assume max have several possible moves.
- Before min decides, there is some element of chance involved (e.g., dice rolled; each outcome with some probability).
  - Chance nodes are added and the outgoing edges labeled with the probabilities.
- Same applies to max nodes further down the tree; i.e.,
  - For each move for min, there are chance nodes added before decisions for max.
- Tree can become large very fast!



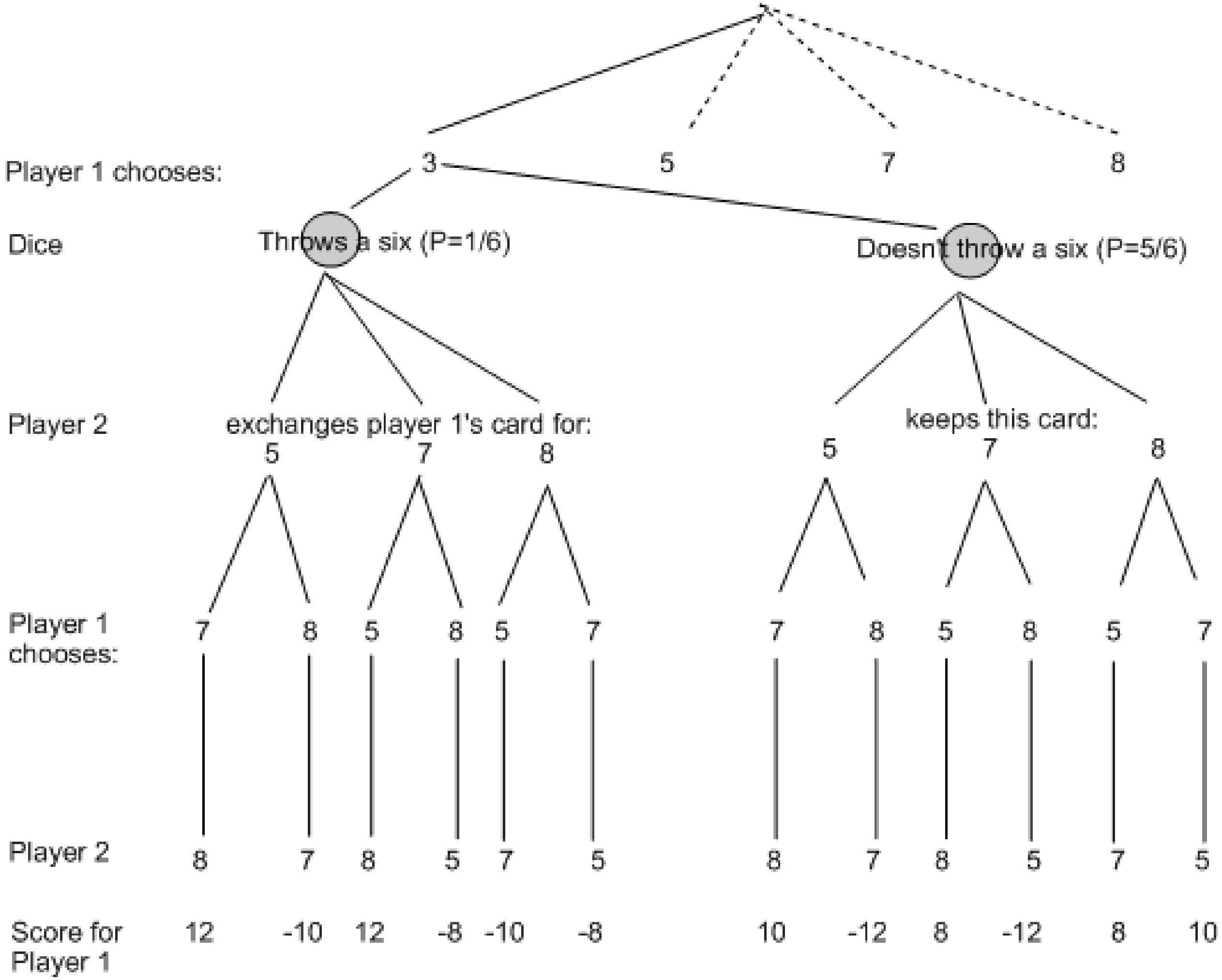
Example of expectiminimax values

- Example below shows calculation of weighted expected values at chance nodes.

- Demonstrates potential problem when payoff function values are skewed;
  - Resulting values at chance nodes are an average (mean);
  - Large skewing in payoff values can cause MAX to make the wrong decision.



Expectimax Diagram



Player 1 chooses: 3 5 7 8

Dice Throws a six (P=1/6) Doesn't throw a six (P=5/6)

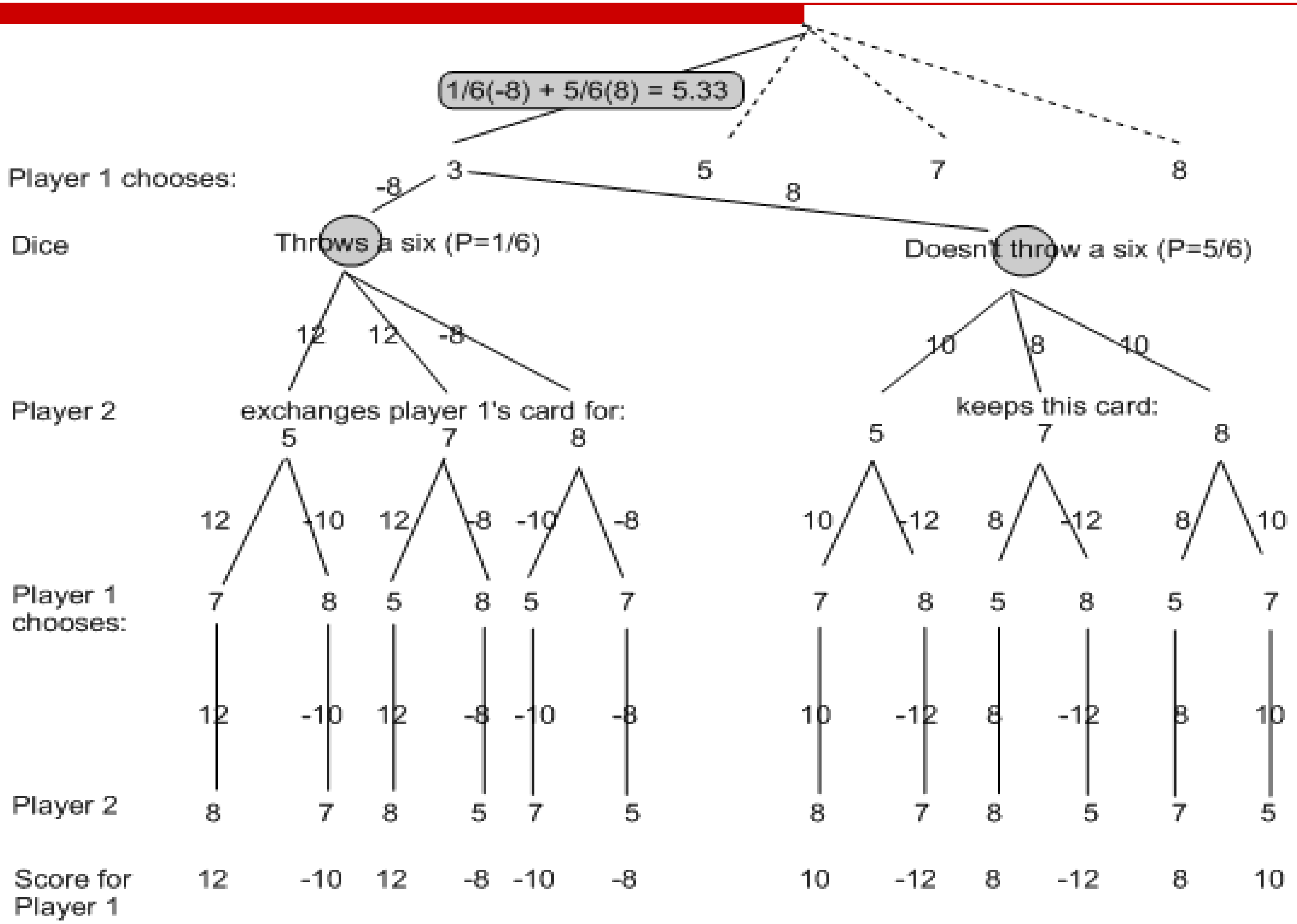
Player 2 exchanges player 1's card for: 5 7 8 keeps this card: 5 7 8

Player 1 chooses: 7 8 5 8 5 7 7 8 5 8 5 7

Player 2 8 7 8 5 7 5 8 7 8 5 7 5

Score for Player 1 12 -10 12 -8 -10 -8 10 -12 8 -12 8 10

Expectimax Calculations



Local Search- Iterative improvement

- Might have a problem in which we only care about the goal itself; path to goal is irrelevant.

- Similarly, might be true that every state can be considered a goal, but has an associated cost.
  - We therefore have a minimization (or maximization) problem.
  - E.g., trying to minimize cost or maximize payoff

- Can often solve such a problem as follows:
  - Start with an initial state/solution and modify the state/solution via actions in order to improve the quality of the state/solution.

- This is known as iterative improvement
  - i.e., working in iterations to get closer to goal

Local search algorithms- iterative improvement

- Previously: systematic exploration of search space.
  - Path to goal is solution to problem
- Yet, for some problems path is irrelevant. In many optimization problems, for example, 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 problem

Local search

- Can often solve such a problem as follows:
  - Start with an initial state/solution and modify the state/solution via actions in order to improve the quality of the state/solution.
- This is known as iterative improvement
  - i.e., working in iterations to get closer to goal

Local search and optimization

- Local search= use single current state and move to neighboring states.
- Advantages:
  - Uses very little memory
  - Find often reasonable solutions in large or infinite state spaces.
- Are also useful for pure optimization problems.
  - Find best state according to some *objective function.*
  - e.g. survival of the fittest as a metaphor for optimization.

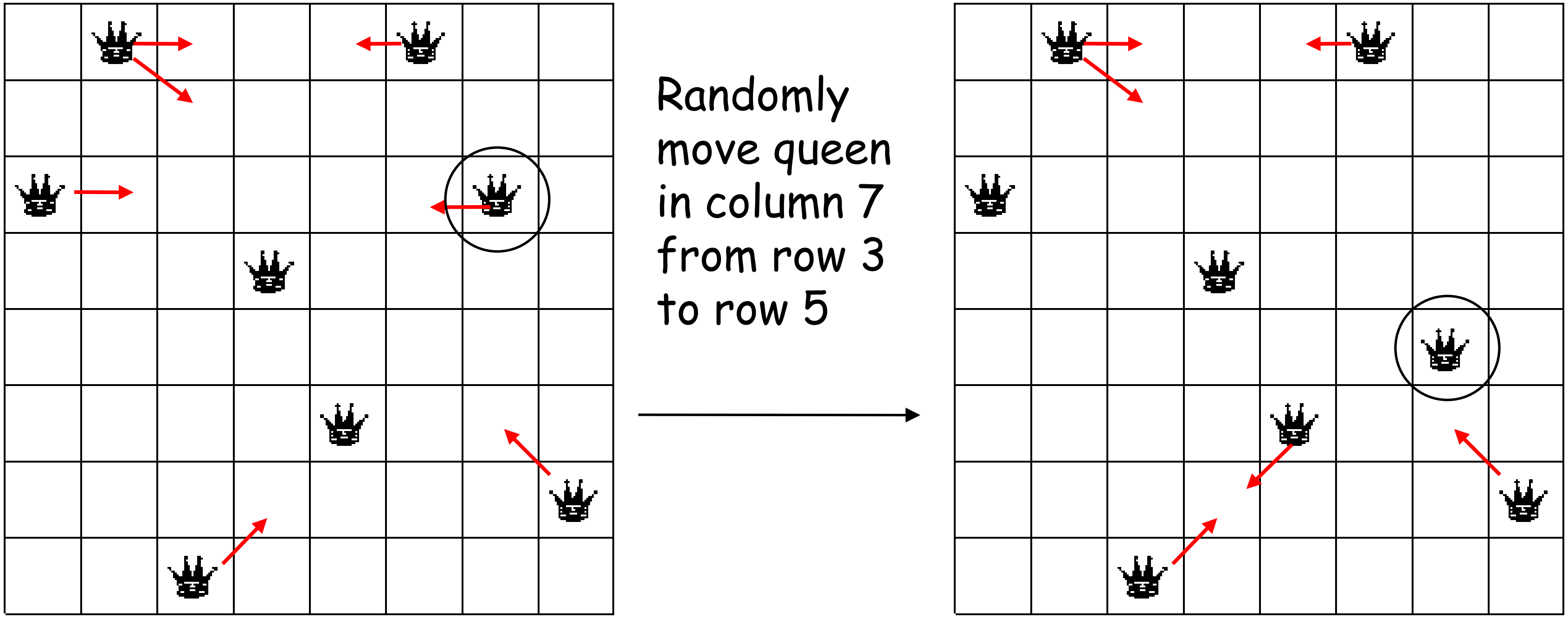
Local search illustrated

- Consider the 8-queens problem; goal is a board configuration w/o any attacks.

- Assume a state representation in which all 8 queens are placed on the board; one queen per column:
  - Some states will be goal states while others are not.
  - Can move from state-to-state by randomly choosing a column and randomly switching the queen in the column to another row.

Local search illustrated

- We need to assign a cost to each configuration;
  - Let the cost be the number of attacks in the configuration.
  - In this case, a goal state would be any arrangement of queens that has a cost of zero.



Randomly move queen in column 7 from row 3 to row 5

Cost = 1+2+1+0+1+1+1+1 = 8

Cost = 0+2+1+0+1+1+0+1 = 6

Elements of Local Search:

- Representation of the solution;
- Evaluation function;
- Neighbourhood function: to define solutions which can be considered close to a given solution. For example:
    - For optimisation of real-valued functions in elementary calculus, for a current solution x_0, neighbourhood is defined as an interval $(x_0 - r, x_0 + r)$.
    - In clustering problem, all the solutions which can be derived from a given solution by moving one customer from one cluster to another;
- Neighbourhood search strategy: random and systematic search;
- Acceptance criterion: first improvement, best improvement, best of non-improving solutions, random criteria;

Some iterative improvement strategies

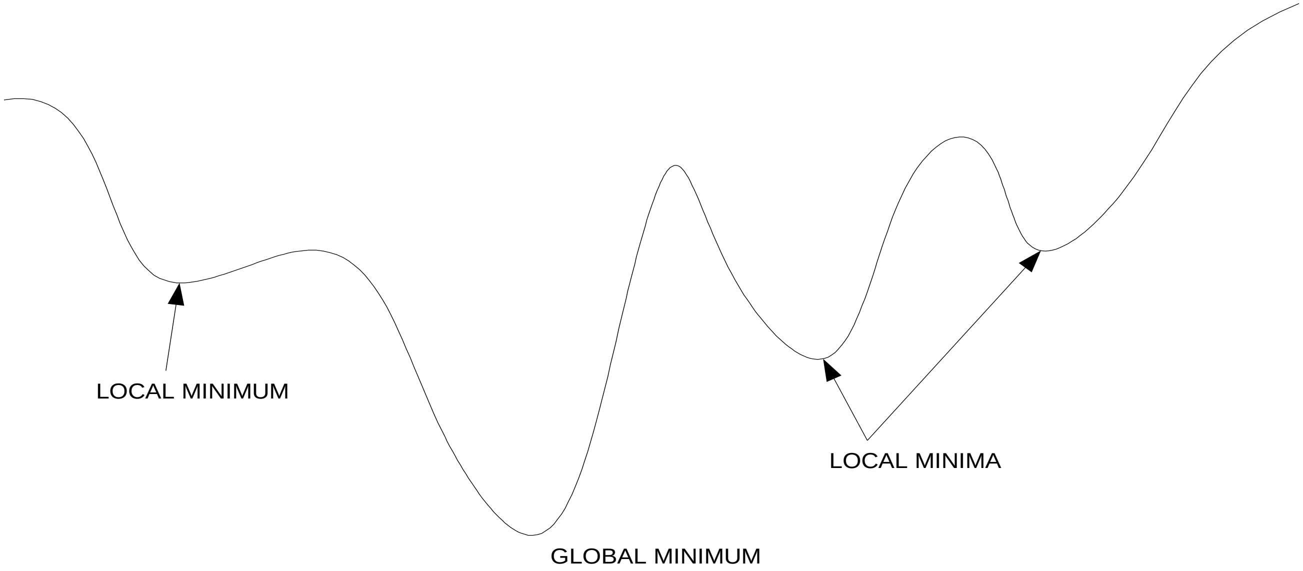
- We will consider several strategies:
  - Hill Climbing (equivalently, gradient descent),
  - Simulated Annealing, and
  - Genetic Algorithms.

- In all cases, we will assume we have some sort of objective function that we would like to either minimize or maximize. (e.g., cost or payoff)

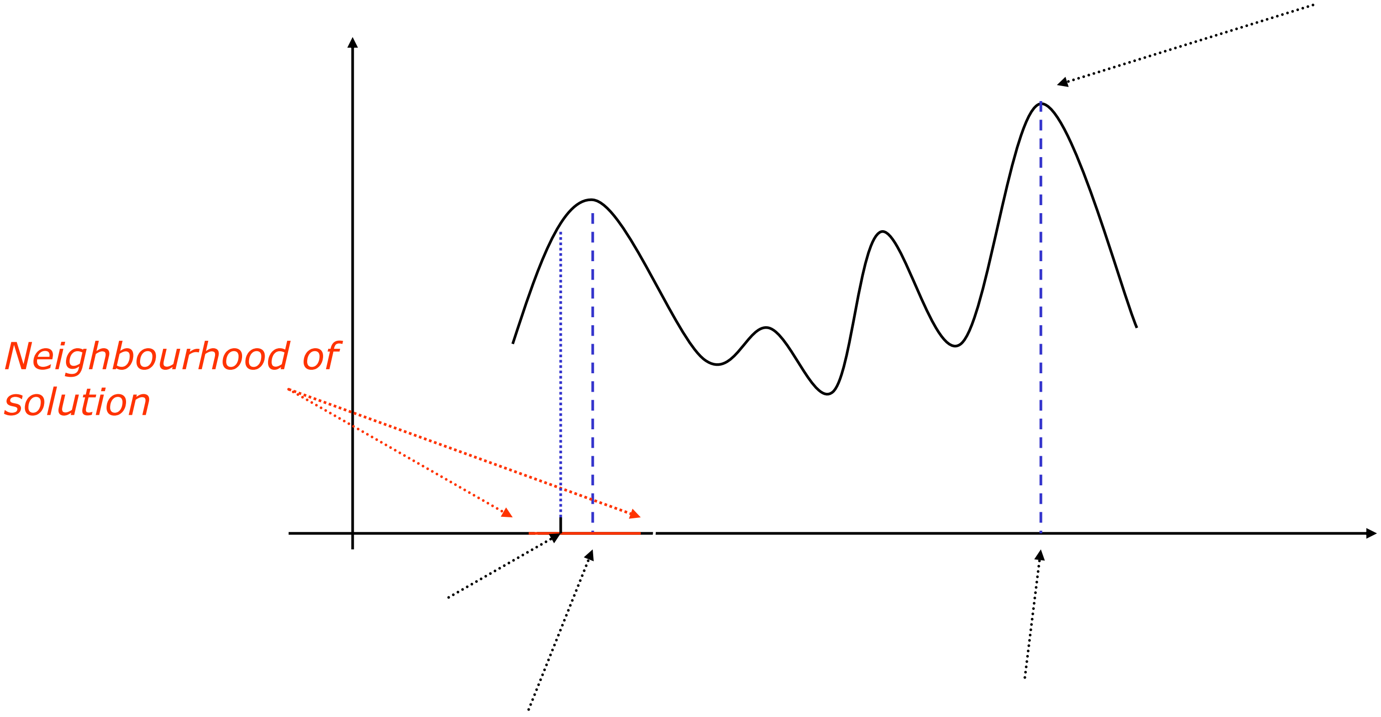
- Note: sometimes when we formulate a problem such that it is solvable via iterative improvement, we need to make a decision about allowing feasible vs. infeasible solutions;
  - If we allow infeasible solutions, we need to augment the objective function with penalties to make infeasible solutions more costly (cf. the 8-queens problem).

Visualization of iterative improvement

- Can visualize iterative improvement via an energy landscape;
  - Each point on landscape corresponds to a solution and has a cost.
- Goal is to find the global minimum (equivalently, maximum); i.e., the deepest (equivalently, highest) cost point.
  - Be careful of local minima (suboptimal solutions).



Optimisation Problems: terminology



Hill climbing (gradient descent)

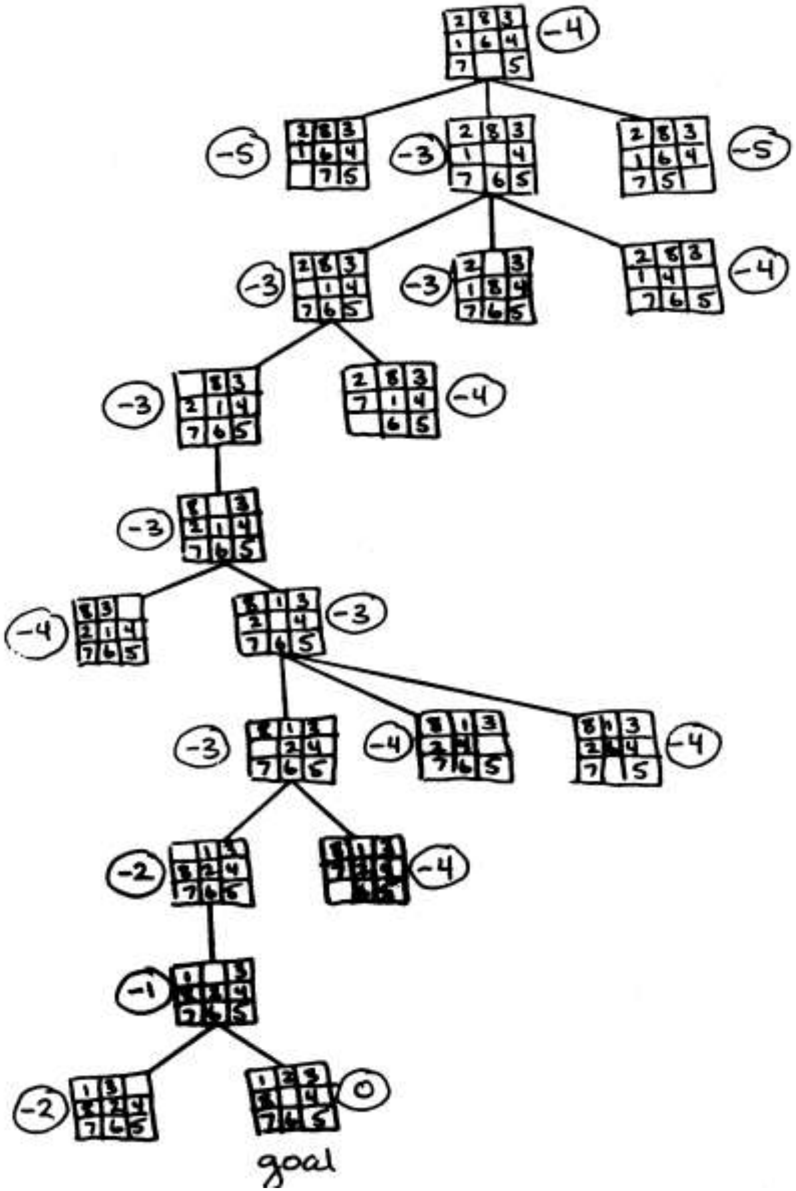
- Simple local optimization strategy;
  - Generate a random initial solution;
  - Pick best possible neighboring solution;
  - Repeat if neighbor has better cost, otherwise break. (Note: maximizing cost here)
- Pseudo-code:

```
curr_state = generate_initial_solution();
while (1) {
    next_state = best_neighbor(curr_state);
    if (cost(next_state) < cost(curr_state)) {
      break; // nb: no longer better to go to a neighbor.
    } else {
      curr_state = next_state;
    }
}
best_state = curr_state;
return best_state;
```

Hill climbing (gradient descent)

- Advantages;
  - No search trees;
  - Typically very fast (if neighbor generation and evaluation is fast).
- Disadvantages:
  - Final solution found subject to initial solution.
  - Can get stuck in a local, sub-optimal solution.
    - Does not look beyond immediate neighbors
    - Does not allow worse moves
  - This method is very much a localized improvement strategy.
- Common to use this approach with either;
  - Many random initial solutions (multi-starts), or
  - Some intelligently constructed initial solution (from another method).

Hill Climbing



- idea: make use of local knowledge to find a global maximum
- variant of DFS, but uses evaluation function
- select the node that looks best in one step
- stop when all next steps are less in value than the current node

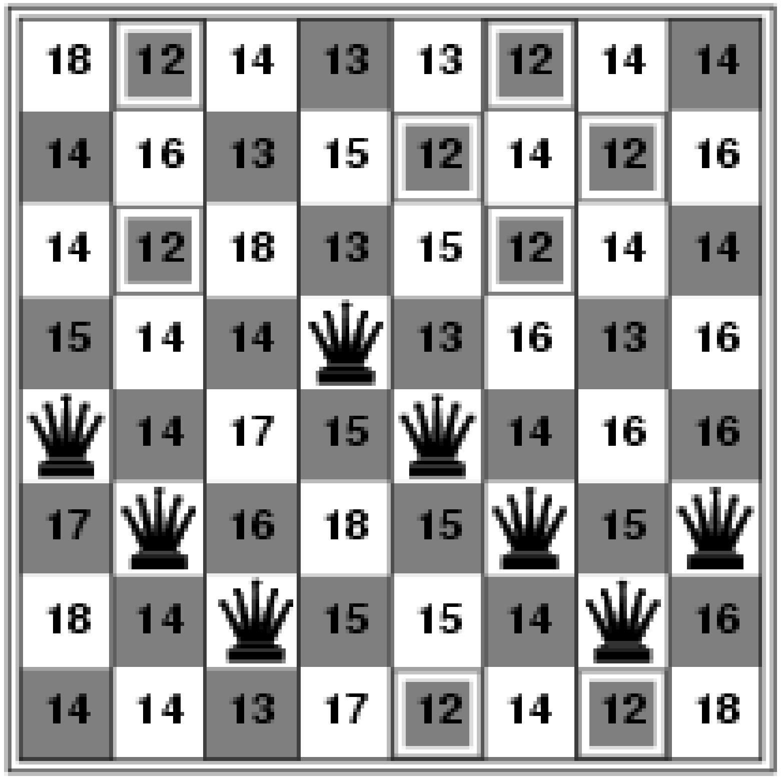
<u>example:</u>

f(S) = - (# misplaced tiles)

Hill-climbing example

- 8-queens problem (complete-state formulation).
- Successor function: move a single queen to another square in the same column.
- Heuristic function *h(n)*: the number of pairs of queens that are attacking each other (directly or indirectly).

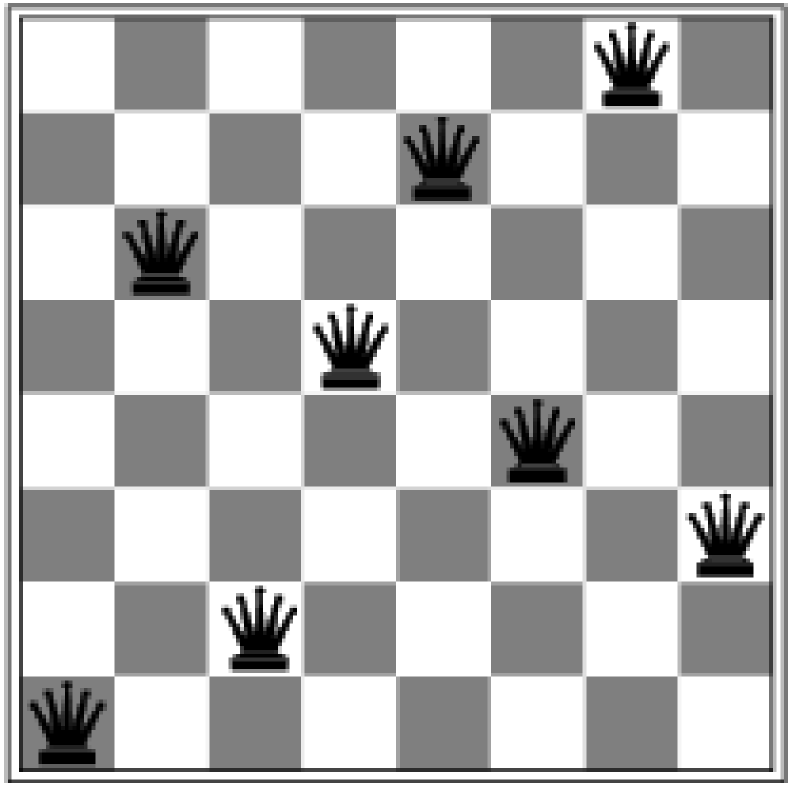
Hill-climbing search: 8-queens problem



18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

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

Hill-climbing search: 8-queens problem



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

Simulated Annealing

- Problem with hill climbing: local best move doesn’t lead to optimal goal
- Solution: allow bad moves
- Simulated annealing is a popular way of doing that
  - Stochastic search method
  - Simulates annealing process in metallurgy

Annealing

- Tempering technique in metallurgy
- Weakness and defects come from atoms of crystals freezing in the wrong place (local optimum)
- Heating to unstuck the atoms (escape local optimum)
- Slow cooling to allow atoms to get to better place (global optimum)

Simulated Annealing

Annealing	Simulated Annealing
Atoms moving	Agent modifying state
towards minimum-energy location in crystal	towards state with global optimal value
while avoiding bad position.	while avoiding local optimum.
Atoms are more likely to move out of a bad position	Agents are more likely to accept bad moves
if the metal's temperature is high.	if the "temperature" control parameter has a high value.

Annealing	Simulated Annealing
The metal’s temperature starts hot,	The “temperature” control parameter starts with a high value,
then it cools off	then it decreases
continuously	incrementally
over time	with each iteration of the search
until the metal is room temperature	until it reaches a pre-set threshold.

Simulated Annealing

- Allow some bad moves
  - Bad enough to get out of local optimum
  - Not so bad as to get out of global optimum
- Probability of accepting bad moves given
  - Badness of the move (i.e. variation in state value ΔV)
  - Temperature T
  - $P = e^{-\Delta V/T}$
- Stochastic search technique

Simulated Annealing

- Generate random initial state and high temperature
- Each iteration
  - Generate and evaluate a random neighbour
  - If neighbour better than current state
    - Accept
  - Else (if neighbour worse than current state)
    - Accept with probability $e^{-\Delta V/T}$
  - Reduce temperature
- End when temperature less than threshold

Simulated Annealing

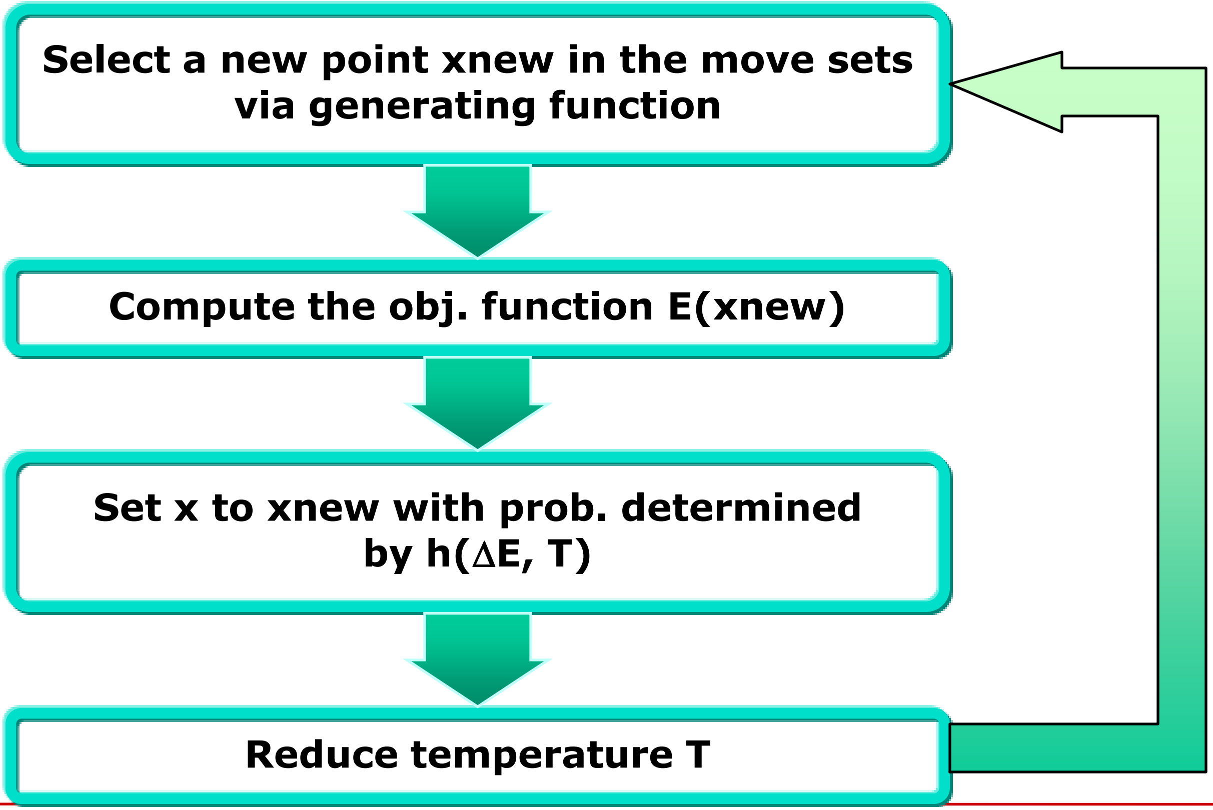
- Advantages
  - Avoids local optima
  - Very good at finding high-quality solutions
  - Very good for hard problems with complex state value functions
- Disadvantage
  - Can be very slow in practice

Simulated Annealing

- Terminology:
  - Objective function $E(x)$: function to be optiimized
  - Move set: set of next points to explore
  - Generating function: to select next point
  - Acceptance function $h(DE, T)$: to determine if the selected point should be accept or not. Usually $h(DE, T) = 1/(1+\exp(DE/(cT)))$.
  - Annealing (cooling) schedule: schedule for reducing the temperature T

Simulated Annealing

- Flowchart:



Select a new point xnew in the move sets via generating function

Compute the obj. function E(xnew)

Set x to xnew with prob. determined by h(ΔE, T)

Reduce temperature T

Accepting moves in simulated annealing

- Accepting bad moves in annealing follows the Metropolis Criterion; an idea from statistical mechanics (mathematical description) of the annealing process.

- Given T, let current solution be **f** and a neighboring solution be **g.**

- If **cost(g) < cost(f)**, accept new solution (improving solution) with probability = 1

- If **cost(g) > cost(f),** accept new solution with probability

  0 < exp(-(cost(g)–cost(f))/T) < 1 (worsening solution).

  - At large T, **exp(-(cost(g)–cost(f))/T)** ≈ 1 and all bad moves accepted;
  - At small T, **exp(-(cost(g)–cost(f))/T)** ≈ 0 and few bad moves are accepted.

- By accepting worsening moves, annealing is performing a type of probabilistic hill-climbing to avoid getting trapped in poor locally minimal solutions.

- We degenerate into gradient descent (local improvement) at low temperatures.

Pseudo-code for simulated annealing

```
bool accept(f, g) {
    delta_cost = cost(g) - cost(f);
    if (delta_cost <= 0) return true;
    return (exp(-delta_cost/T) > random_number(0,1));
}

void simulated_annealing() {
    T = initial_temperature();
    f = initial_solution();
    do {
        for( move = 1; move <= move_limit; move++ ) {
            g = neighbor_solution(f);
            if (accept(f,g)) f = g;
        }
        T = reduce_temperature(T);
    } while (!stop(T));
}
```

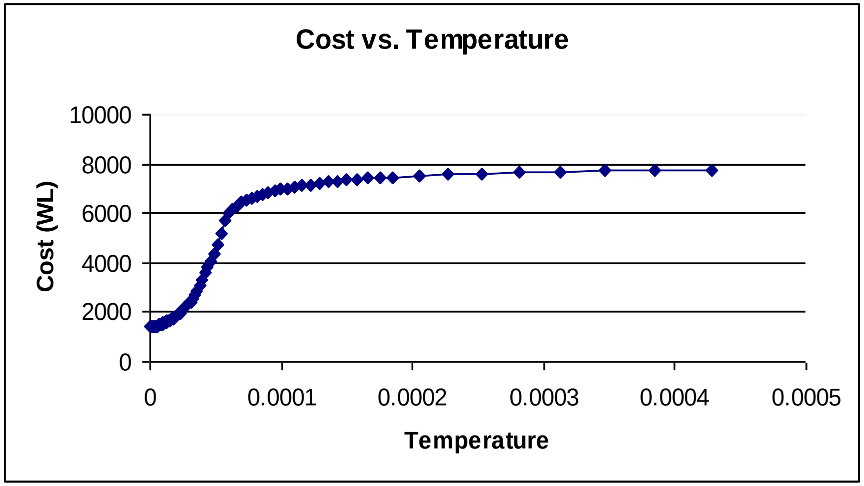
- Start with a random initial solution and a high temperature.
- Slowly reduce the temperature until it is less than some small value.
- Perform large number of moves (actions to change configuration) at each temperature.
- Accept new solutions according to some rule.

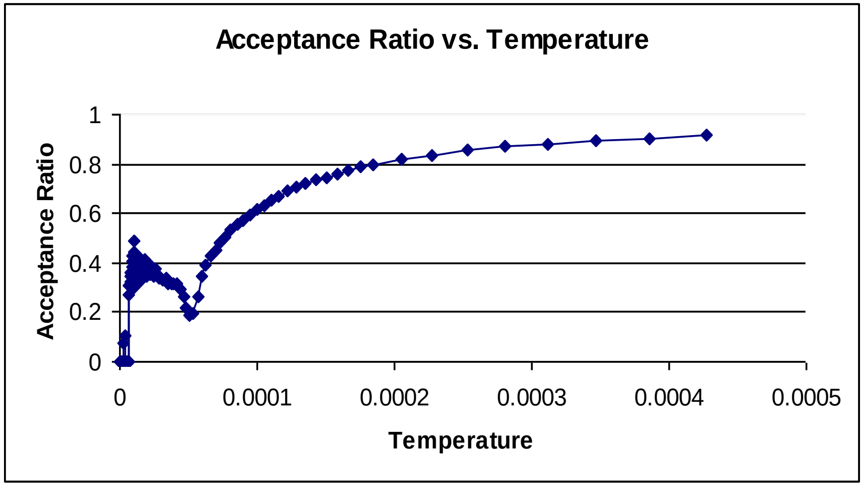
- Note that we accept both good and bad moves (bad moves accepted with probability which is a function of T).

Illustration of cost and acceptance during simulated annealing

- Define the acceptance rate as the number of moves accepted at any given temperature divided by the total number of moves attempted.

- As T → 0, cost decreases (approaching a minimum); acceptance rate also decreases as there are fewer good moves to make and the probability of accepting a bad move decreases.





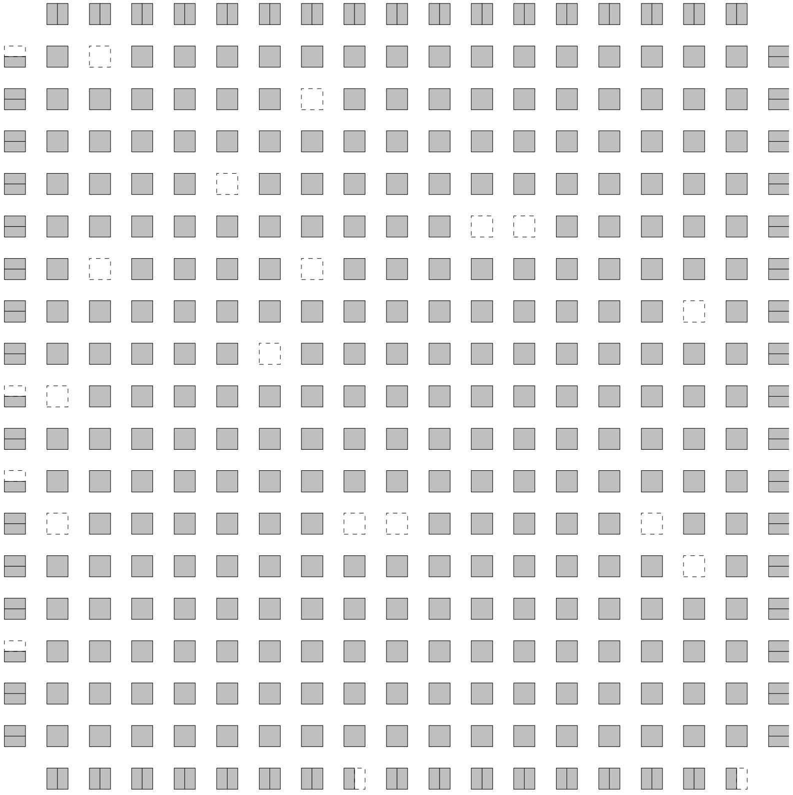
Simulated annealing

- Advantages:
  - Avoids local minima since worsening moves allowed.
  - Popular algorithm for hard problems with complicated cost functions.
  - Works really well at finding good, high quality solutions.
  - There is a theoretical proof of convergence (asymptotic), so somewhat useless in practice.

- Disadvantages:
  - Can be very slow in practice since lots of moves are performed.

Illustration of simulated annealing for fpga placement

- Want to place blocks of logic onto a 2-dimensional grid to minimize wire length (for example).

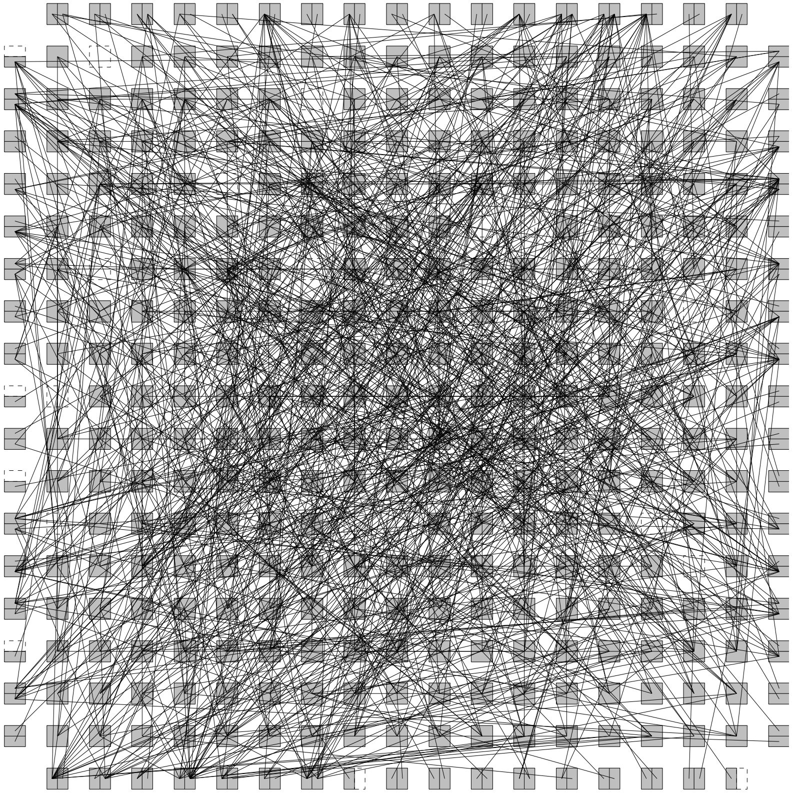


Initial Placement. BB Cost: 74.6951

- Run a simulated annealing algorithm;
  - Start with an random initial placement of logic blocks onto locations in the 2-dimensional grid.
  - Configurations changed by randomly swapping pairs of logic blocks.
  - Cost of a placement will be the total wire length to connect the logic blocks together.

- Plot shows an initial random placement of logic blocks (not too interesting).

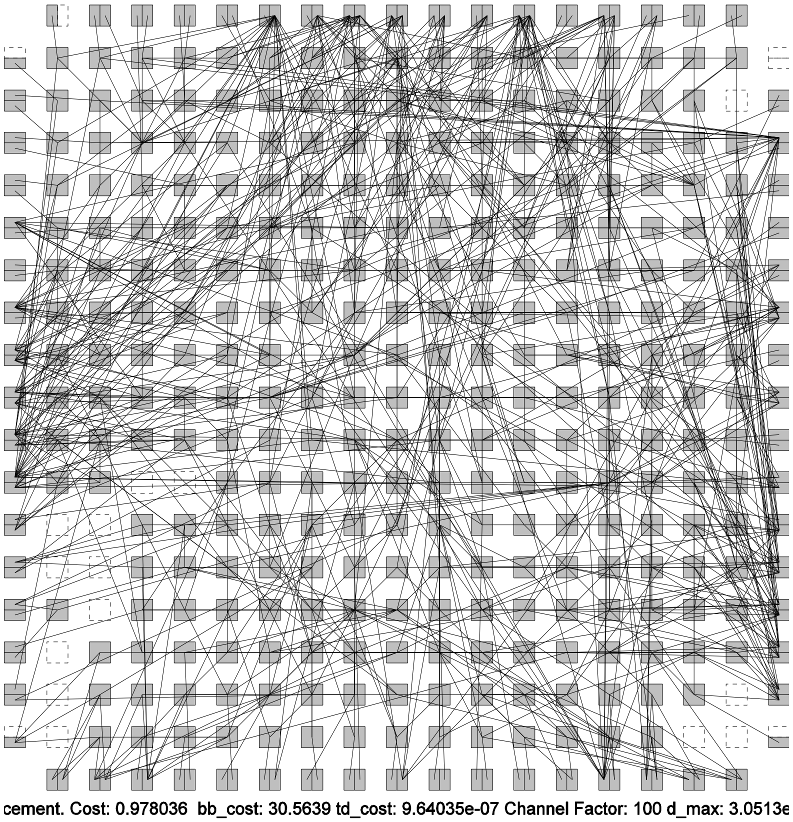
Illustration of simulated annealing for fpga placement



Initial Placement. BB Cost: 74.6951

- Plot shows all the edges connecting different pieces of logic.
- Diagram is very cluttered.

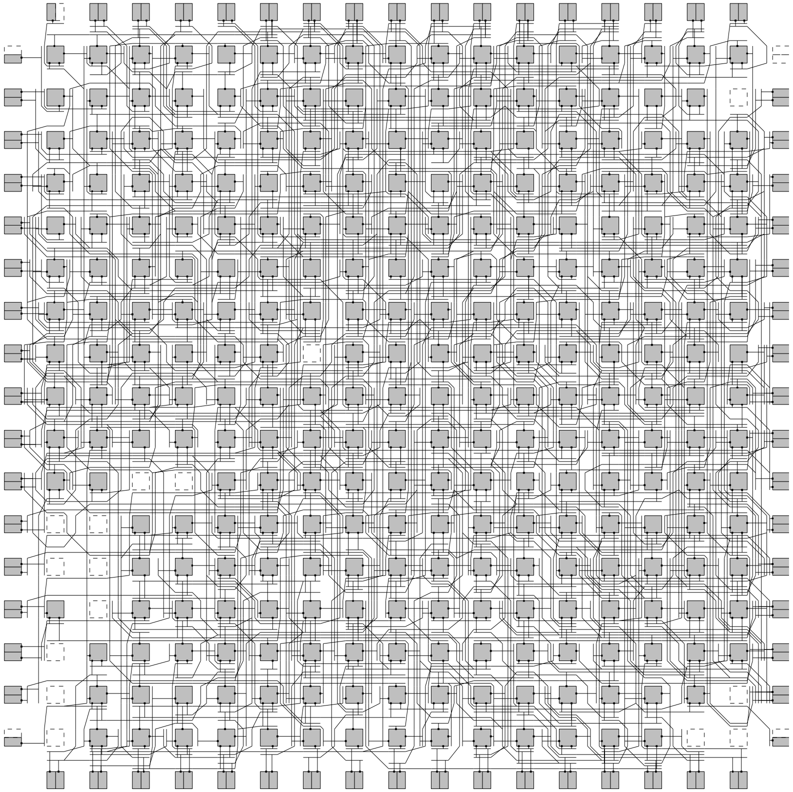
Illustration of simulated annealing for fpga placement



cement. Cost: 0.978036 bb_cost: 30.5639 td_cost: 9.64035e-07 Channel Factor: 100 d_max: 3.0513e

- Plot shows all the edges connecting different pieces of logic after simulated annealing has been applied.
- Diagram is not nearly as cluttered.

Illustration of simulated annealing for fpga placement



Routing succeeded with a channel width factor of 8.

- Plot shows all the edges after routing (using A* search!).
- Routing the edges around the logic blocks was much easier due to the use of simulated annealing to provide an optimized placement.

Evolutionary Computing

- Refers to a family of computing (optimization) techniques which adopt the idea of smart natural adaptation (evolution).

- The main idea was introduced in 1968 and 1975 by John Holland.

- Natural evolution: The widely accepted theory of natural creation and evolution (Charles Darwin – 1858)

- According to Darwin, the average features (Genes) of living species **evolve (become more adapted to the environment)** through **generations** (**evolution cycles**)

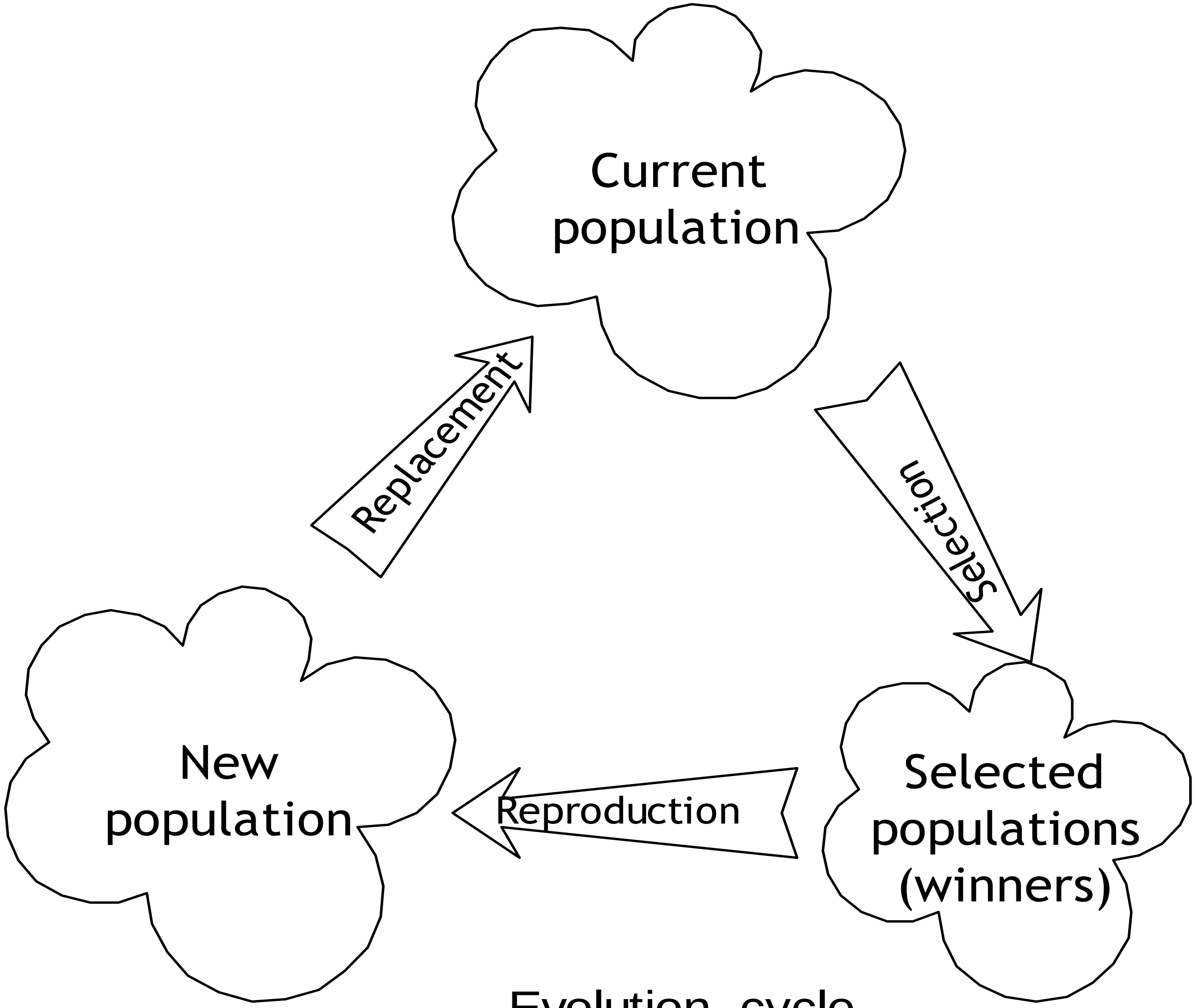
- **Evolution cycle** includes **3 phases**:

  - selection,
  - reproduction and,
  - replacement

Evolutionary Computing

- **Selection**:
  - Members of a population compete for survival and mating.
- **Reproduction:**
  - Winners mate and produce new generation (**diversity is important**).
- **Replacement:**
  - New generation completely (or partially) replaces the older generation.
- The idea of natural evolution is adopted to solve variety of search (optimization) problems when the search space is too large (NP-Complete).
- Evolutionary inspired algorithms for solving search problems are ***Genetic Algorithms.***

Evolutionary Computing



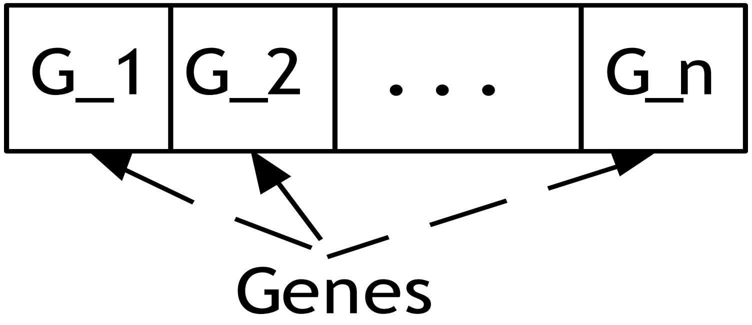
Evolution cycle

Genetic Algorithm Preliminaries (Fitness Functions)

- Some preliminary steps are required.
- Need to be able to evaluate the **fitness** of members of the population. Can use a **fitness function**.
    - The fitness function measures the quality of a problem solution (bigger fitness is better).
- Hence, when we talk about survival, mating, etc. the idea is that new generations are (overall) **more fit** than older generations.

Genetic Algorithm Preliminaries (Chromosome Encoding)

- Some preliminary steps are required.
- Need to store, or **encode**, potential solutions. We can represent potential problems solutions as in **chromosome form**.
  - We borrow another concept (i.e., the concept of a chromosome) from evolution.
- **Chromosomes commonly encoded as bitstrings … Possibly something else.**



Genetic Algorithms (Main Algorithmic Steps)

- Given a problem, determine **encoding** and **fitness function**.
- Form an initial population (either randomly or uniformly) – i.e., construct a set of possible problem solutions. The initial population becomes the current population.
- Subsequently, perform the evolutionary cycle :
  - Select a small subset of the current population based on a **specific selection rule**.
  - Reproduce to generate the next population based on a **specific reproduction rules** (the size of next population must be the same as size of current population).
  - Replace the current population according to a **specific replacement rule**.
  - Terminate if the **optimization goal** is achieved or the computing time is over. Otherwise go to step 1.

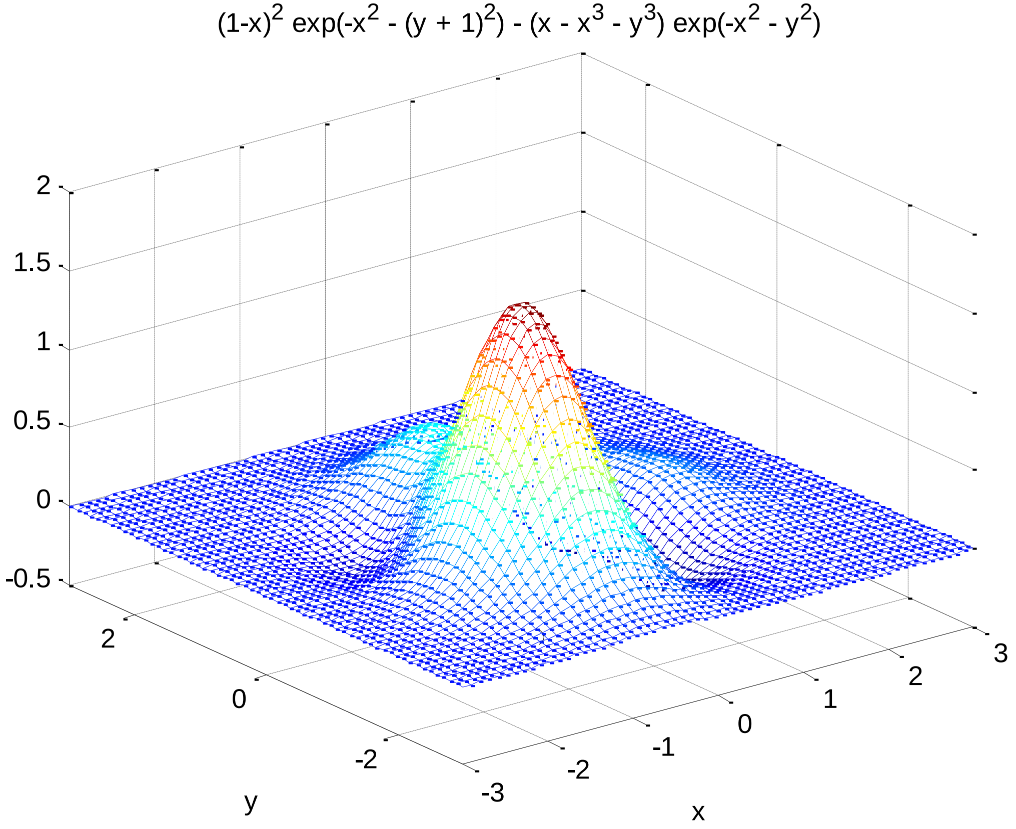
Example Problem (1 of 7)

- Consider the following maximization problem:

$$f(x,y) = (1-x)e^{-x^2-(y+1)^2} - (x - x^3 - y^3)e^{-x^2-y^2}$$

$$(-3 \le x \le 3)$$

$$(-3 \le y \le 3)$$



Example Problem (2 of 7)

- Need to pick a fitness function; use the function itself (in this example):

$$f(x,y) = (1-x)e^{-x^2-(y+1)^2} - (x - x^3 - y^3)e^{-x^2-y^2}$$

$$(-3 \le x \le 3)$$

$$(-3 \le y \le 3)$$

  - **Note:** actual function will not normally be available, so a heuristic fitness function may be necessary.

- Need to pick an encoding. Use bit-strings to encode the (x,y) value:

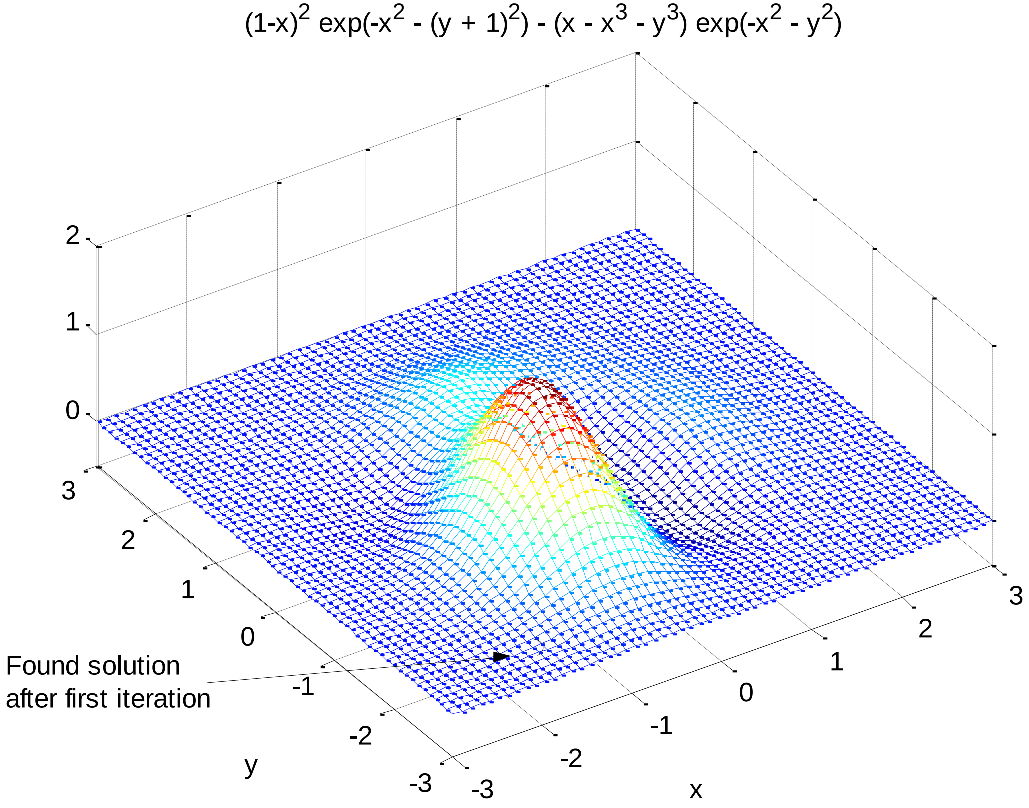
$$\underbrace{11010101}_{x}\underbrace{01011010}_{y}$$

Example Problem (3 of 7)

- Other details (some alluded to earlier; some to still talk about):
  - Population size is 5 (i.e., 5 solutions maintained).
  - Selection of individuals for mating is proportional. (more later)
  - Reproduction uses crossover and mutation (with probability 0.05). (more later)
  - Replacement of population is complete (i.e., old generation is completely replaced with new generation – no survivors).

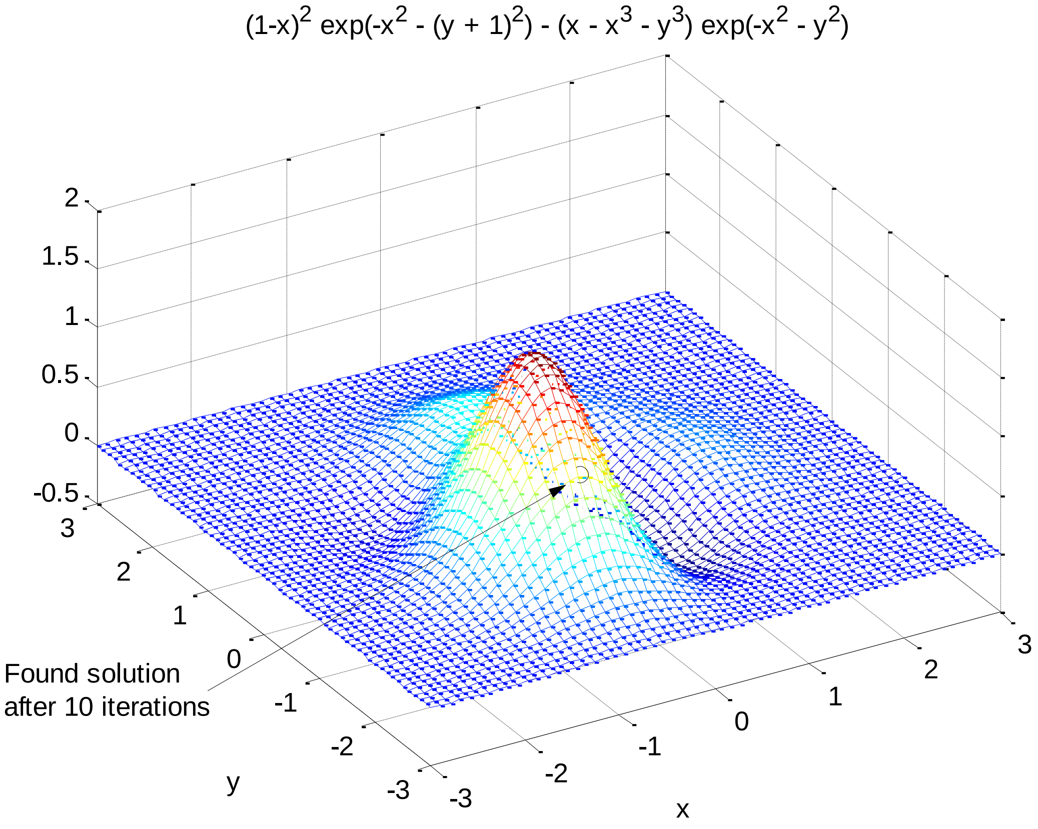
Example Problem (4 of 7) – Progression

- View of solution found after 1-st generation:



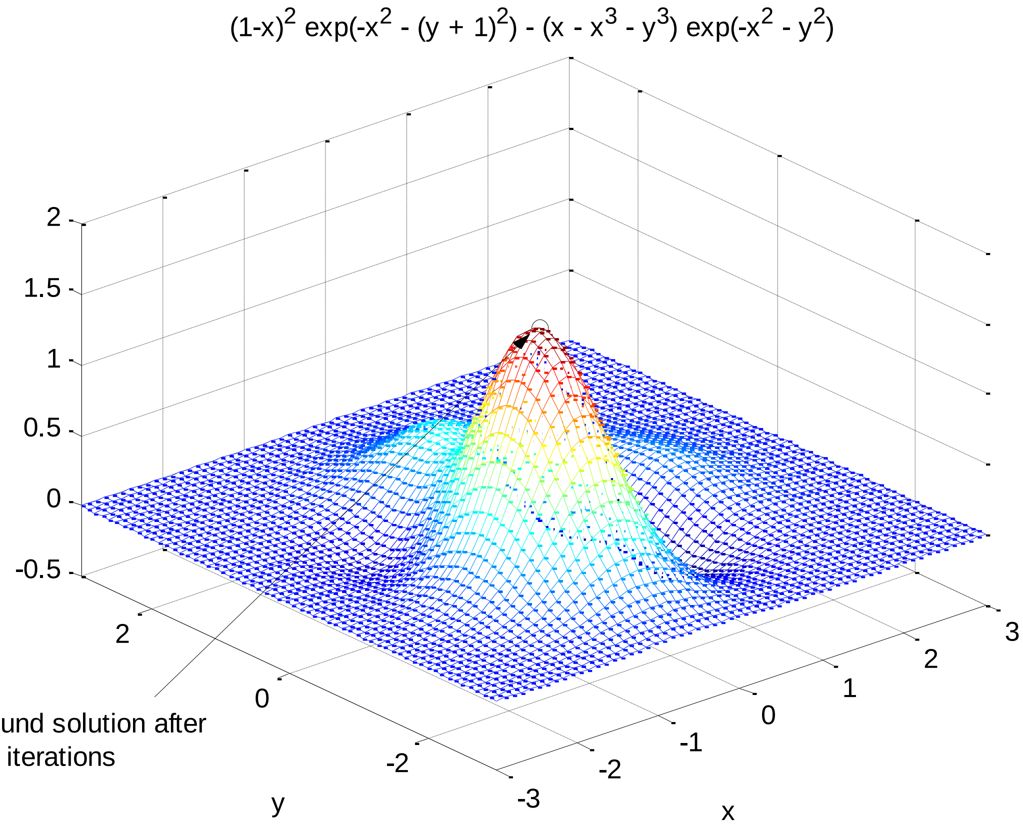
Example Problem (5 of 7) – Progression

- View of solution found after 10-th generation:



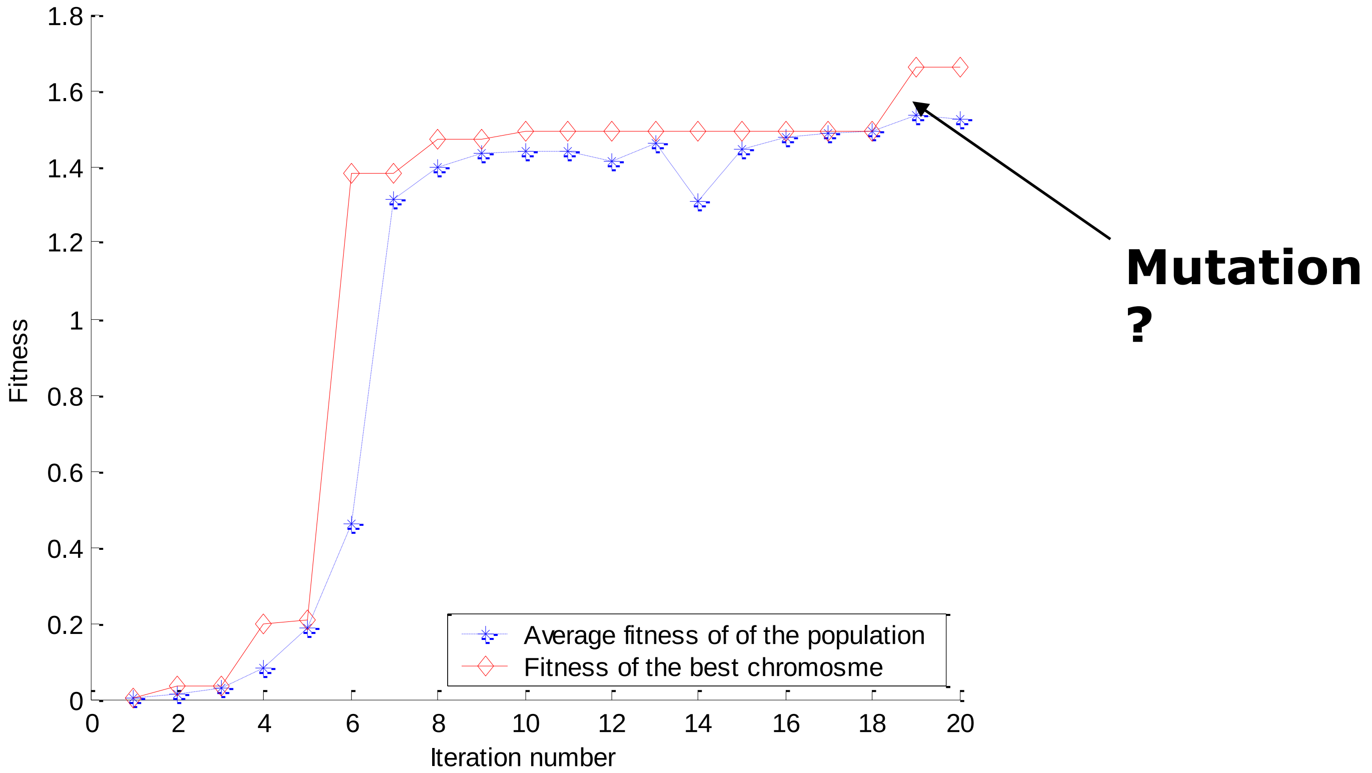
Example Problem (6 of 7) – Progression

- View of solution found after 20-th generation:



Example Problem (7 of 7)

- Can view progression of the algorithm over the evolutionary cycles (iteration).
- Note the evolution in both average and best chromosome.



Selection Rules (1 of 2)

- We need to pick members of the current population for reproduction (all members of population contest for reproduction).

- A selection rule basically is a competition rule which gives **more chance to the chromosomes with better fitness**.

- Chance of each chromosome (member) being selected depends on its fitness. This dependency could be:

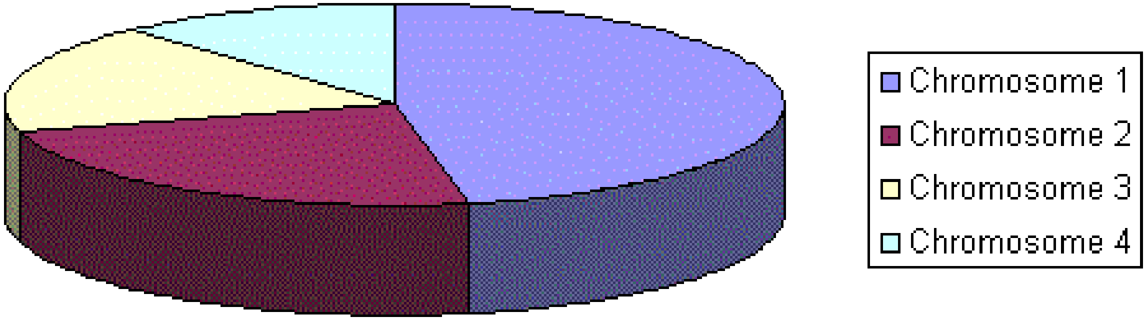
  - Proportional
  - Exponential (greedy – emphasizes higher fitness)

- Having calculated the chance of selecting each chromosome, a roulette wheel scheme could be used to select required number of selected generations.

Selection Rules (2 of 2)*

- Illustration… The "piece of pie" is somehow proportional to the fitness of the particular chromosome.

- Roulette wheel will randomly select a chromosome.



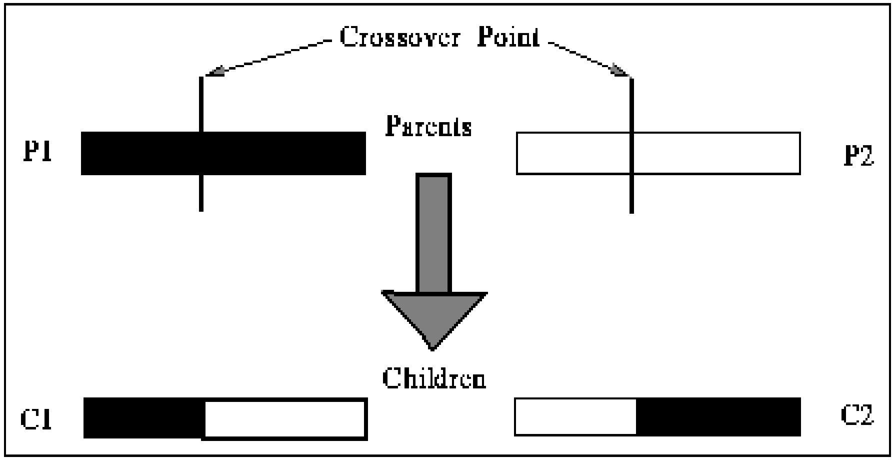
Reproduction Rules (1 of 3)

- Once we have selected two members, we need to use them for reproduction.
- Generally, a good reproduction rule should create good **DIVERSITY** in the new population.
  - Idea is to help improve the exploration of the search space.
- Well known techniques for reproduction (to achieve diversity):
  - Crossover.
  - Mutation.

Reproduction Rules (2 of 3) - Crossover

- Crossover is a simple method to take two members of the current population and produce two children.

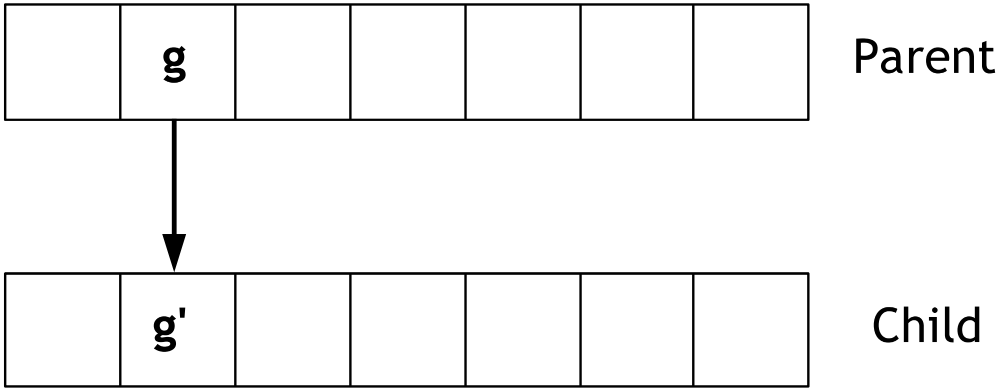
  - E.g., using chromosome representation, select at random a fixed position in the chromosome; children receive one part from each parent.



- Hopefully, children will exhibit "good properties" (inherent good qualities) of their two parents.
  - **Note:** with small populations, diversity is removed rather quickly.

Reproduction Rules (3 of 3) – Mutation*

- Mutation adds some additional randomness into the process with some small probability (e.g. less than 0.1).
  - Every gene of the parent chromosome has chance of mutation.
  - Adds diversity to the population.



$$g' = \begin{cases} g & \text{with prob } (1 - p) \\ \text{some other value,} & \text{with prob } p \end{cases}$$

Replacement Rules

- Must decide (given current population and produced children), how to form the new population.

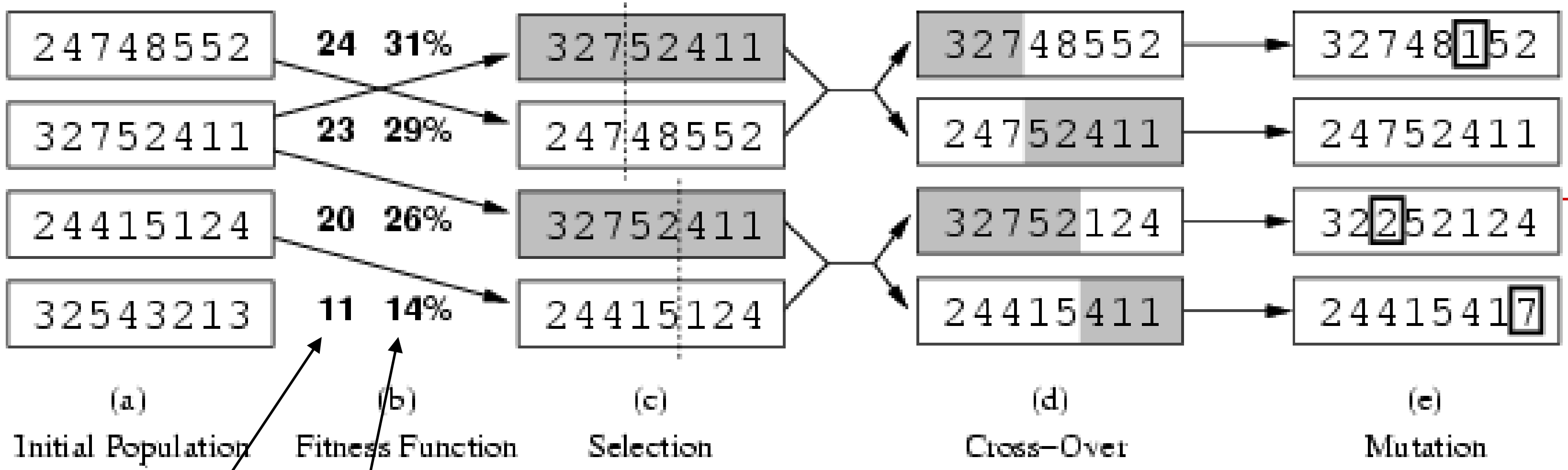
- In most existing variants of GA, the number of individuals in the next population is the same as current population.

- Different replacement schemes:

  - Whole replacement of current population with the reproduced population.

  - Individuals with the best fitness from the current and reproduced population are chosen to make new population (greedy approach).

- Being greedy (can) compromise the exploration capability of the algorithm (premature convergence).

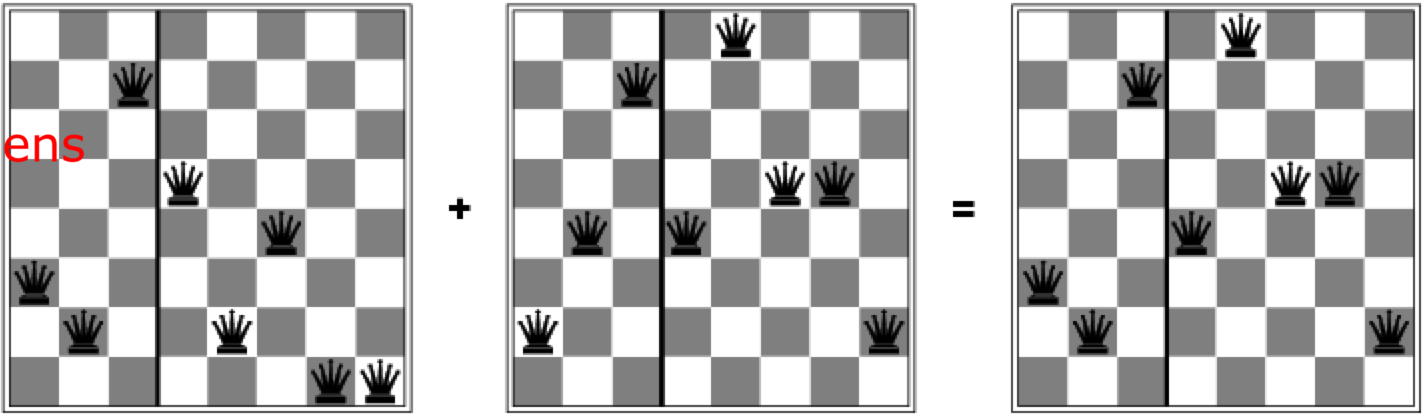


fitness:
#non-attacking queens

- □ Fitness fun
  queens (min

probability of being
regenerated
in next generation

- □ 24/(24+23+20+11) = 31%
- □ 23/(24+23+20+11) = 29% etc



Genetic Algorithm Termination

- When should a Genetic Algorithm terminate?
  - Approaches optimum solution with probability of 1, assuming infinite number of iterations (generations).
  - Problem dependent, but typically stop when (i) a certain performance level is met or (ii) available computational effort is over.

Genetic Algorithm Problems

- Genetic algorithms are not always the right approach.

- Method is very domain dependent (i.e., depends on nature of the problem), and generic packages fail in many cases.

- Constrained optimization problems can not be solved easily; hard (for instance) to pick suitable crossover, mutation operators, etc.

  - E.g., crossover and mutation might continually yield infeasible solutions that require much correction.

- Could be time consuming and is not good for fast algorithms.

Genetic Algorithms Applications

- Generally speaking is good for **offline** complex optimization problems.
- Offline clustering problems in different areas (VLSI circuit partitioning, Software re-engineering)
- Offline signal processing (image processing).
- Planning problems.
- Power distribution systems.

Final Note on GAs

- Can see parallels with simulated annealing (SA):

  i. Large steps typically happen early on, while population still diverse.
     - cf. high temperature phase in SA.

  - Smaller steps as diversity is “bred out.”
     - cf. cooling in SA.

  - Possibility of an unexpected move, even in later generations, via mutation.
     - Helps get out of local minima.
     - cf. probability of worsening move in SA.