EC504 ALGORITHMS AND DATA STRUCTURES
FALL 2020 MONDAY & WEDNESDAY
2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Hw 8: Due tonight

Project: Due Friday 4/30

① Report
+ email to dac@bu.edu
and toslali@bu.edu

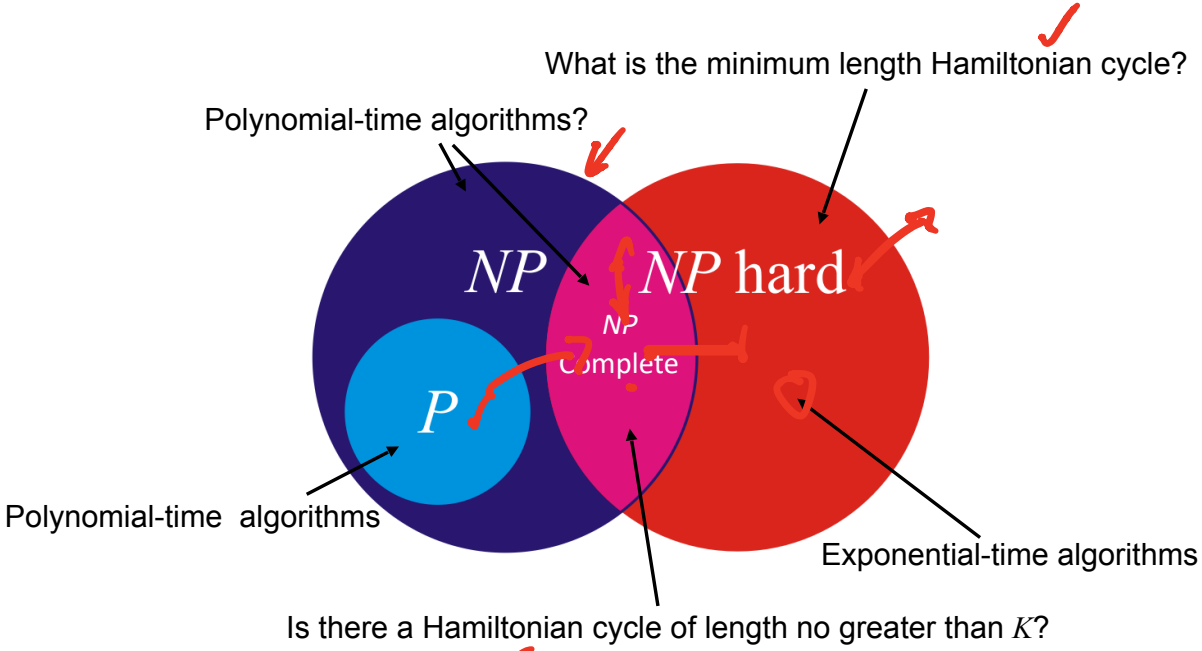
② Code: make directory
FinalProject in one
team member site, upload
to SCC

→ Put location of directory
in e-mail with report.

Include instructions on how
to build + Run...

NP and NP Hard

Consider this Venn diagram where $P \subseteq NP$



$A \leq_p B$

Any A → Some B..

Garey & Johson → 300 pgs of examples of NP-Com.

TSP Approximations

- Dynamic programming (Held-Karp, Bellman): $O(n^2 2^n)$ exact algorithm
- Restrict to Metric (or Euclidean) TSP: assume distances $d(i, j)$ satisfy the following
  - Symmetry: $d(i, j) = d(j, i)$
  - Triangle inequality: $d(i, j) \leq d(i, k) + d(k, j)$ for all i, j, k
- Approximate algorithm
  - Find a minimum spanning tree G = (V, T)
  - Add a duplicate edge to each edge in the minimum spanning tree T, with same distance, resulting in augmented edges T'
  - Find an Euler tour in G' = (V, T'). Note one exists because degree of each vertex is even.
  - List the vertices in order visited by Euler tour; skip vertices that appear more than once

TSP Approximations

- Theorem: Approximate algorithm tour has distance no longer than twice the distance D* of the optimal TSP tour
  - Proof: Let D = weight of minimum spanning tree, D' the weight of the approximate tour

    Then, $D \leq D^*$, as the minimum spanning tree reaches every vertex, and has less constraints than the TSP tour

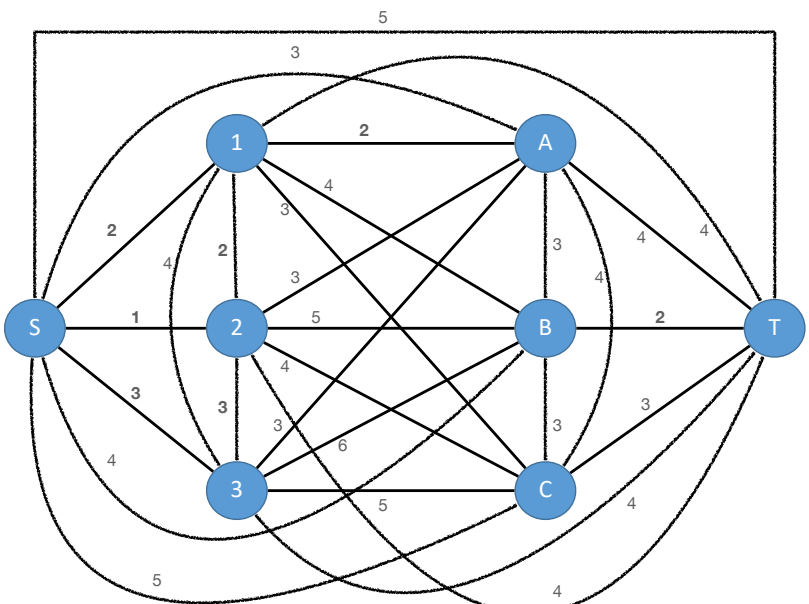
    Also, note the length of the Euler tour is 2D, as it travels every edge of the MST twice

    From the metric property, cutting the corners by deleting the duplicate vertices cannot increase the length,

    Hence $D \leq D^* \leq D' \leq 2D$

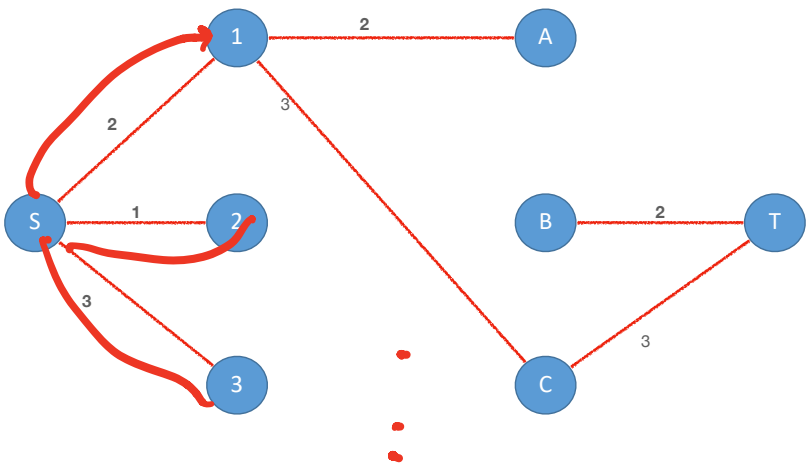
Example

- Find MST: Weight 16



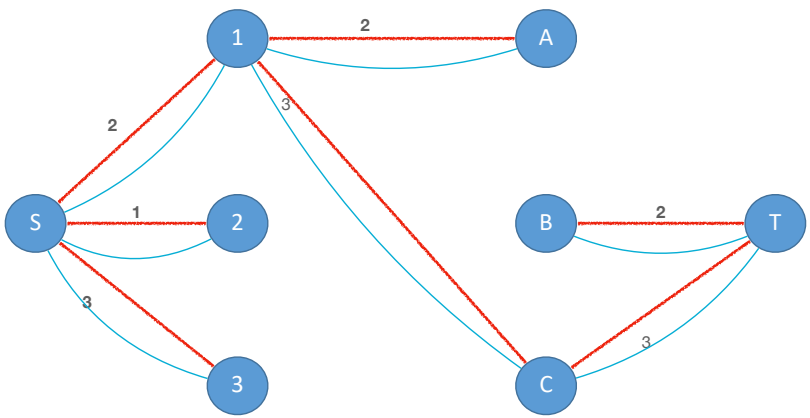
Example - 2

- Double edges



Example - 3

- Find Euler Tour: S-1-A-1-C-T-B-T-C-1-S-2-S-3-S. Length 32

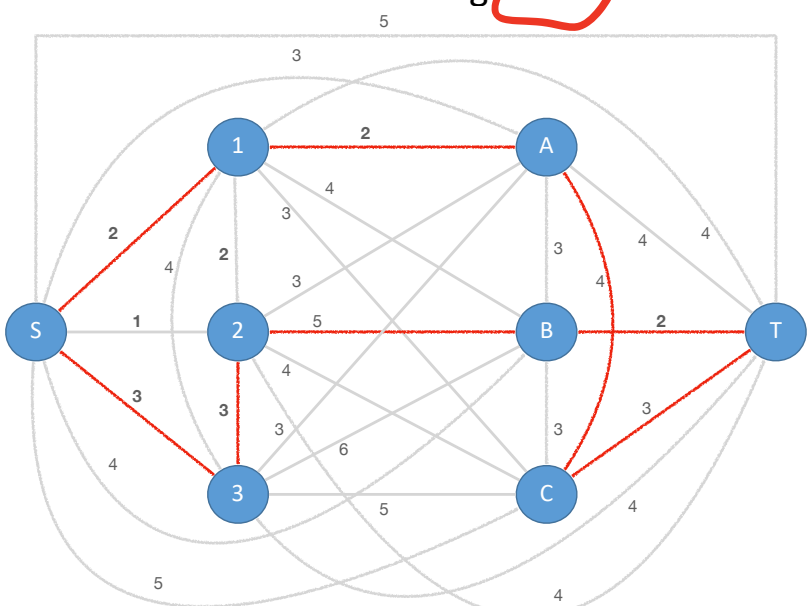


- Cut corners: S-1-A-C-T-B-2-3-S. Length

Example - 3

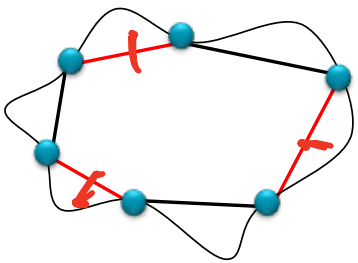
- Euler tour length: 32
- Cut corners: S-1-A-C-T-B-2-3-S. Length 24

$16 \leq D^* \leq 24$



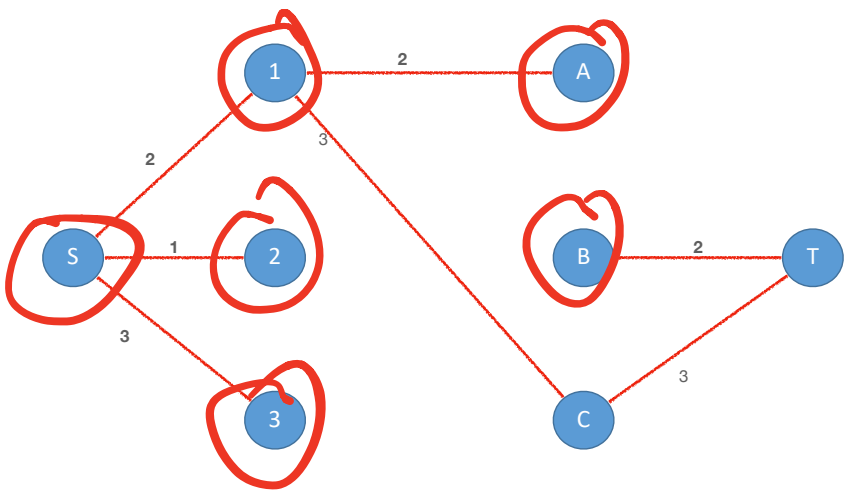
TSP Approximations

- Better approximation: Christofides' algorithm
  - Suboptimality factor 1.5 for Euclidean TSP
  - Key idea: find a minimum weight matching M between all vertices of odd degree in minimum spanning tree T
    - Can be done in polynomial time by algorithm beyond our scope
  - Note: There must be an even number of odd degree vertices
    - Property of undirected graphs: sum of degrees is even
  - The graph $G(V, T \cup M)$ is Eulerian: every vertex has even degree
  - Can find Euler tour in graph
  - Using the same shortcut ideas, can get the 1.5 bound!



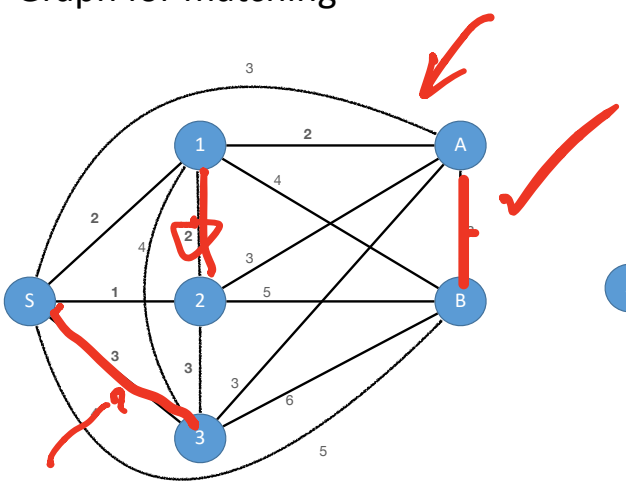
Example - Christofides Approximation

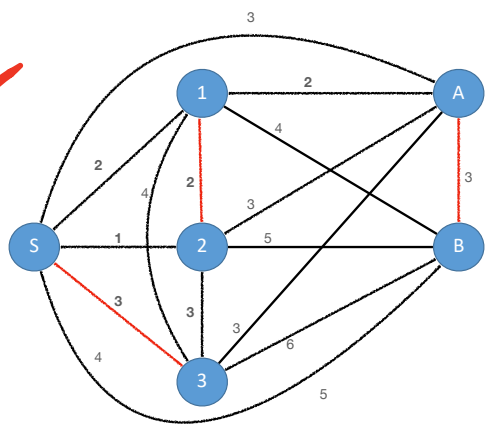
- Vertices S, 1, 2, 3, A, B have odd degree



Example - Christofides 2

- Graph for matching

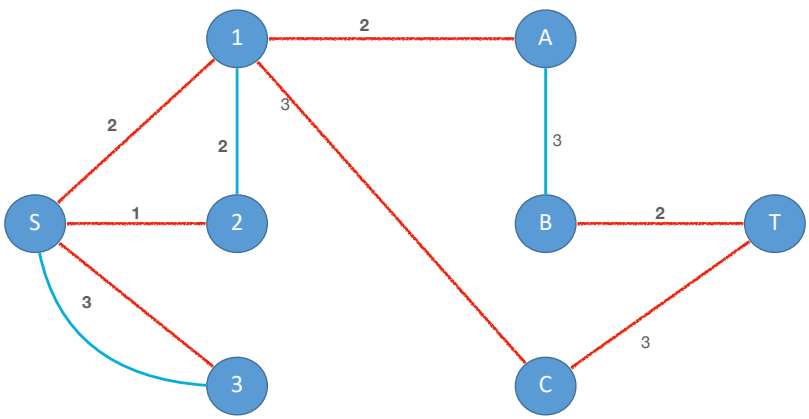




Jack Edmonds → { Paths, Blossoms + Flowers ...

Example - Christofides 3

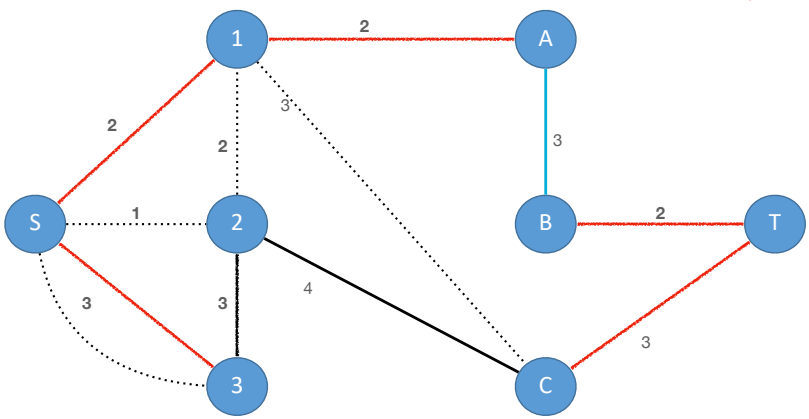
- Eulerian Network



- Euler Tour: S-1-A-B-T-C-1-2-S-3-S: length 24

Example - Christofides 3

- Euler Tour: S-1-A-B-T-C-1-2-3-3-S: Length 24
- Cut corners: S-1-A-B-T-C-2-3-S: Length 22



Knapsack: Polynomial Time Approximation Scheme

- Knapsack: tasks $i = 1, \ldots, n$ with values v_i, weights w_i, total capacity W
- Pseudopolynomial complexity: O(nW)
  - Using Dynamic Programming with iteration in capacity and number of tasks consideredKnow pseudo-polynomial in C using dynamic programming
- Assume $w_i \leq W,\ i = 1, \ldots, n$
- A different dynamic programming algorithm (in Value)
  - OPT(i, v) = min weight of a knapsack for which we can obtain a solution of value ≥ v using a subset of items 1,..., i
  - Optimal value is the largest value v such that OPT(n, v) ≤ W
  - DP computes optimal value in $O(n^2 \max v_i)$ because max value is $O(n \max v_i)$

$$OPT(i, v) = \begin{cases} 0 & v \leq 0; \\ \infty & i = 0,\ v > 0; \\ \min(OPT(i-1, v), w_i + OPT(i-1, v-v_i)) & \text{otherwise} \end{cases}$$

Knapsack: PTAS

- Knapsack Approximation: round up values
  - $\varepsilon \in (0,1]$ is precision parameter
  - Scaling parameter $\theta = \varepsilon v_{max}/(2n)$
  - Rounded values $\bar{v}_i = \lceil \frac{v_i}{\theta} \rceil \theta$
  - Compressed values $\hat{v}_i = \lceil \frac{v_i}{\theta} \rceil$
  - Optimal solutions are the same for rounded and compressed values
  - But compressed values have much smaller range!

Item	Value	Comp.	Rounded	Weight
1	934221	2	1000000	2
2	595634	2	1000000	3
3	1781001	4	2000000	1
4	2345741	5	2500000	3
5	5000000	10	5000000	4

$\varepsilon = 1,\ \theta = 500000$

Knapsack: PTAS

- Theorem: If S is solution to compressed problem, and S^* is any other feasible solution, then $(1+\varepsilon)\sum_{i\in S} v_i \geq \sum_{i\in S^*} v_i$
  - Proof:
    $$\sum_{i\in S^*} v_i \leq \sum_{i\in S^*} \bar{v}_i \leq \sum_{i\in S} \bar{v}_i \leq \sum_{i\in S} (v_i+\theta) \leq \sum_{i\in S} v_i + n\theta$$
  - If $S^* = \{i_{v_{max}}\}$ then $n\theta = \varepsilon v_{max}/2$;
    $$\sum_{i\in S^*} v_i = v_{max} \leq \sum_{i\in S} v_i + \varepsilon v_m ax/2 \leq \sum_{i\in S} v_i + v_m ax/2 \Rightarrow v_{max} \leq 2\sum_{i\in S} v_i$$
  - Thus, $n\theta \leq \varepsilon \sum_{i\in S^*} v_i$ which shows $(1+\varepsilon)\sum_{i\in S} v_i \geq \sum_{i\in S^*} v_i$

Knapsack: PTAS

- Theorem: for any $\varepsilon > 0$, the rounding algorithm computes a solution with value within a $(1+\varepsilon)$ of the optimal solution in $O(n^3/\varepsilon)$ time
  - Proof:
    - Running time is $O(n^2 \hat{v}_{max})$, and $\hat{v}_{max} = \lceil \frac{v_{max}}{\theta} \rceil = \lceil \frac{2n}{\varepsilon} \rceil$

Further Improvements on NP-Hard Approximations

- Approximate algorithms for NP-Hard problems are fast, but often have significant gaps in performance relative to optimal algorithms
- Can often improve on approximate algorithms
- Local Search
  - Starts with an initial solution given by a fast algorithm
  - Searches a "neighborhood" of the initial solution for improved solutions
  - Fast, but may not find optimal
- Branch and Bound
  - Similar in spirit to A* search
  - Intelligent enumeration of search space, using bounds to eliminate regions to search
  - Can be slow

Local Search

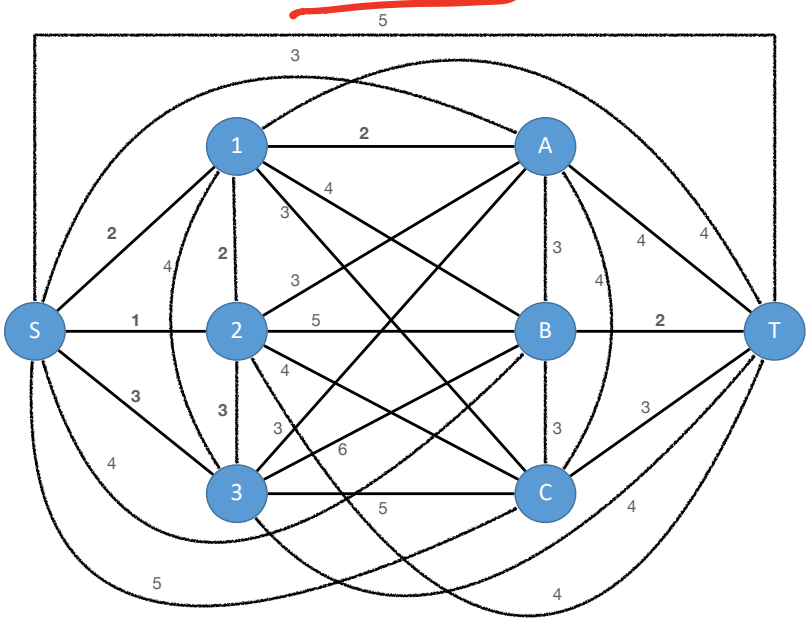
- Assume we have a candidate solution x to an optimization problem
  - Value function V(x) which we seek to maximize
- Define a neighborhood of x: N(x) to be solutions easily obtained from x
  - Easy = in linear or polynomial time
  - Ideally, N(x) is not very large
- Local search algorithm:
  - While there exists $y \in N(x)$ such that $V(y) > V(x)$,
    - Find $y \in N(x)$ such that $V(y) > V(x)$
    - Set x = y
- Algorithm terminates with a solution at least as good as initial solution x with limited extra computation

Example: Local Search for TSP

- Assume we have an initial tour x = $v_1, v_2, \ldots, v_n$
- Neighborhood N(x): any tour where the position of one vertex is changed and inserted in a different position:
  - e.g. y = $v_1, v_n, v_2 \ldots, v_{n-1}$ where v_n was inserted into a different order
- Algorithm:
  - Compute an initial tour x, and set it to current best tour x.
  - Search neighborhood N(x) for shorter tour y
    - If y is found, switch current best tour x = y
    - Repeat search
  - If no shorter tour is found in N(x), return x
- Many variations possible by changing neighborhood (e.g. swap pairs, …)
- May converge to local minimum, not global

Example

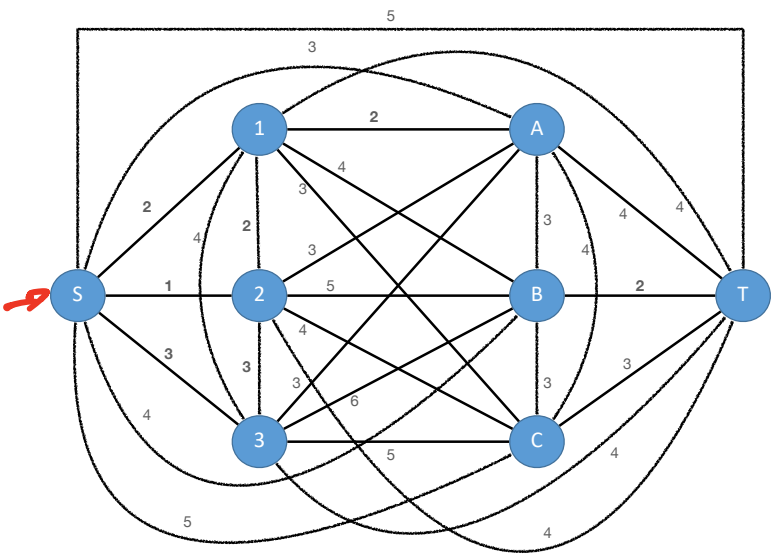
- Initial tour: Greedy S-2-1-A-B-T-C-3-S, Length 24
- Initial tour: Christofides, S-1-A-B-T-C-2-3-S, length 22



Example

- Initial tour x: MST tour S-1-A-C-T-B-2-3-S, Length 24
  - y = S-2-1-A-C-T-B-3-S Length 23, swap
  - y = S-2-1-A-B-C-T-3-S Length 21, swap
  - No further improvements possible

- Initial tour x: Christofides tour
  - x = S-1-A-B-T-C-2-3-S , Length 22
  - y = S-2-1-A-B-T-C-3-S , Length 21: swap
  - No further improvements possible



Branch and Bound

- Similar in spirit to A* search, but more general
- Start with a problem with a finite number of candidate solutions. Examples of such problems will be described later. The goal of this problem is to find a solution of minimum value, given a value function
- Describe partial solutions to the problem as nodes in a graph, with arcs connecting partial solutions when one can evolve from another. That is, think of nodes in this graph as representing partial solutions, and graphs as feasible transitions to other solutions.
- For each node in this graph, there is a function which computes the value of the solution which best completes the current partial solution. In dynamic programming on graphs, we call this the cost-to-go. However, computing this function is a nontrivial endeavor, and it is very hard to do for NP-Hard problems. Instead, we assume we have an approximation to the cost-to-go, usually an optimistic lower bound which is easy to compute.
- Search the graph of possible solutions, in a greedy manner, giving priority to nodes that have the lowest estimates of costs to go. Note that these estimates change as one explores the graph. The search continues until a complete solution is found, and the cost of that solution is at least as good as the cost of other solutions, based on the lower bound estimates.

Example: Integer Knapsack

- We construct the graph of partial solutions. As nodes, we denote subsets of the set {1,…,n}, including the empty set, which fit in W. The empty set is the starting node in our graph. Note that this is a conceptual set of nodes; we don't want to enumerate all of these nodes, as it could be enormous. Rather, we want to find the node which has the optimal solution without examining most of the other nodes…
- As arcs, we include a directed arc from node a to b if node b can be reached from node a by adding a new task. This implicitly assumes that node b can be executed in less than or equal to W. The weight of the arc (a,b) is the value of the task which was added
- For each node a, define the bound function b(a) as follows: Let W(a) be the residual time left after scheduling the tasks in a. Let S(a) be the set of tasks not in a. Then, b(a) is the solution of the fractional knapsack problem (allowing partial credit) for scheduling S(a) in T(a) time. Note that b(a) is an optimistic bound,

Example: Integer Knapsack

- Notation: tasks V, indexed by k = 1, …, N, with weights w_k, values v_k, total capacity W

  Partial solution X_n is subset of tasks that fit W. That is, $\sum_{j \in S_n} w_j \leq W$

  Bound $B(X_n)$ is an optimistic estimate of how much additional value can be added to X_n that fit in W

  Computed by solving fractional knapsack with tasks $V - X_n$, capacity $W - \sum_{j \in X_n} w_j$

  Initially, $X_0 = \varnothing$, with estimate value $D(X_0) = B(X_0)$

  Add X_0 to priority min-queue, with key value $D(X_0)$

Example: Integer Knapsack

\

- Notation: tasks k = 1, …, N, with weights w_k, values v_k, total capacity W
- Partial solution X_n is subset of tasks that fit W. That is, $\sum_{j \in S_n} w_j \leq W$
- Bound $B(X_n)$ is an optimistic estimate of how much additional value can be added to X_n that fit in W
  - Computed by solving fractional knapsack with tasks

- n0 be the empty set, and compute b(n0), and D(n0) = V(n0)+ b(n0), and pred(n0) = 0. Add n0 to the search list.
- Find the element n in the search list which has maximum D(n).
- If there are no n1 which can be reached from n, set b(n) = 0, update D(n) = V(n) and reinsert n into search list.
- Else, for each n1 which can be reached from n0, if V(n)+ w(n,n1) < V(n1), update V(n1) = V(n)+ w(n,n1), where w(a,b) is the weight of arc (a,b). Insert n1 into the search list. If b(n1) is -1, compute b(n1). Update D(n1) = V(n1) + b(n1), and remove n from search list.
- Repeat b) until node n is reached for which there are no outgoing arcs remaining and b(n) is already set to 0.

- To show that the algorithm is correct, note the termination condition: You reach a node for which no additional tasks can be scheduled, and for which the actual distance D(n) is equal to the actual value scheduled, and for which the value D(n) is greater than or equal to an upper bound estimate of the value which can be scheduled from all other partial schedules reachable from the initial node.
-

Zero-Sum Games and adversarial Search: Minimax, Alpha-Beta pruning

John Von Neumann
Games + the Theory of
Economic Behavior → 1944.

Two-player games

The object of a search is to find a path from the starting position to a goal position

In a *puzzle-type* problem, you (the searcher) get to choose every move

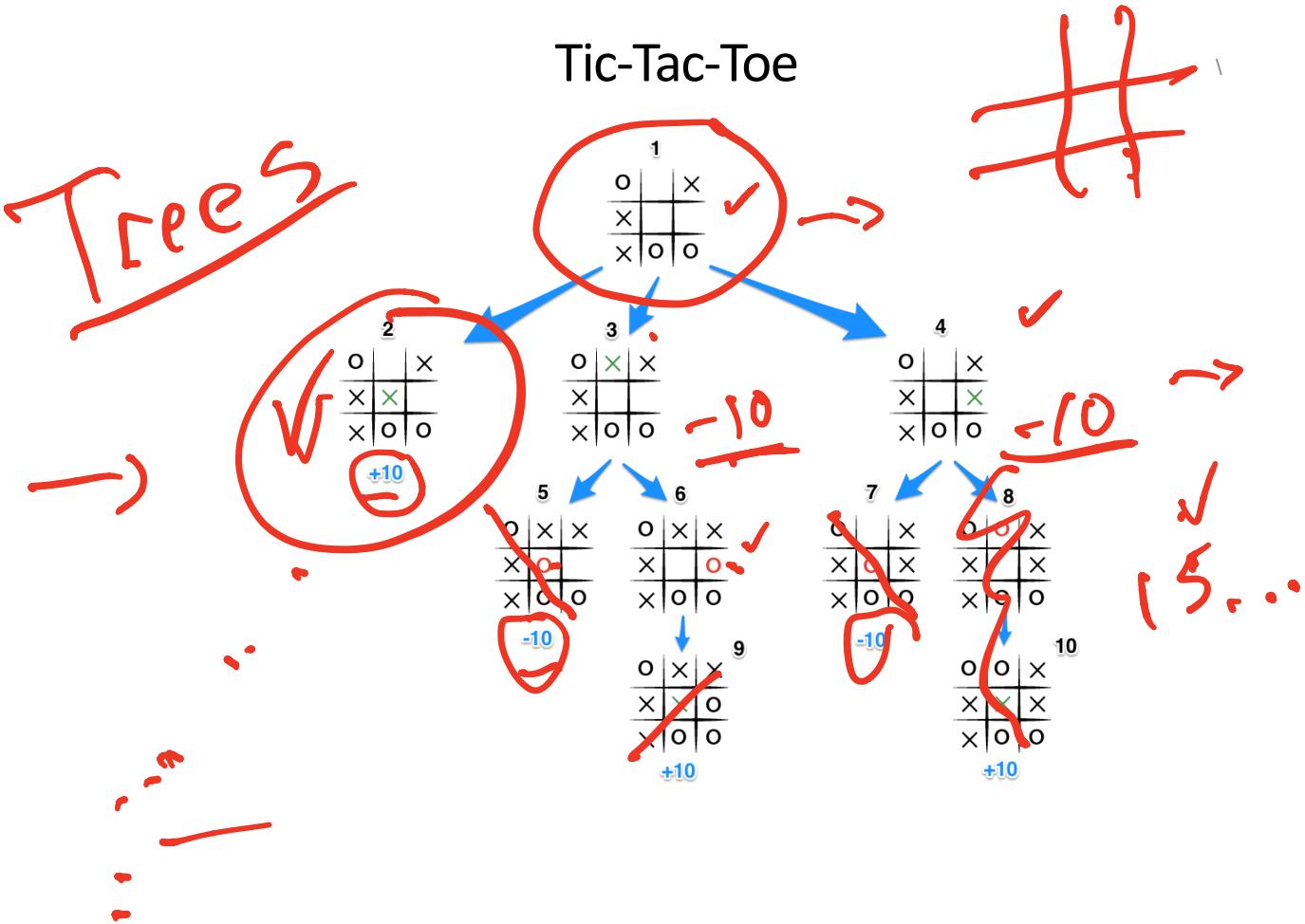
In a two-player competitive game, you alternate moves with the other player

The other player doesn't want to reach *your* goal

Your search technique must be very different

Types of games

Tic-Tac-Toe



\

Payoffs

Each game outcome has a *payoff,* which we can represent as a number

By convention, we prefer positive numbers

In some games, the outcome is either a simple win (+1) or a simple loss (-1)

In some games, you might also *tie,* or *draw* (0)

In other games, outcomes may be other numbers (say, the amount of money you win at Poker)

John Nash → Nash Equilibrium

Zero-sum games

Many common games are zero-sum: What I win ($12), plus what you win (-$12), equals zero

Not all games are zero-sum games

For now, we consider only zero-sum games

From our point of view, positive numbers are good, negative numbers are bad

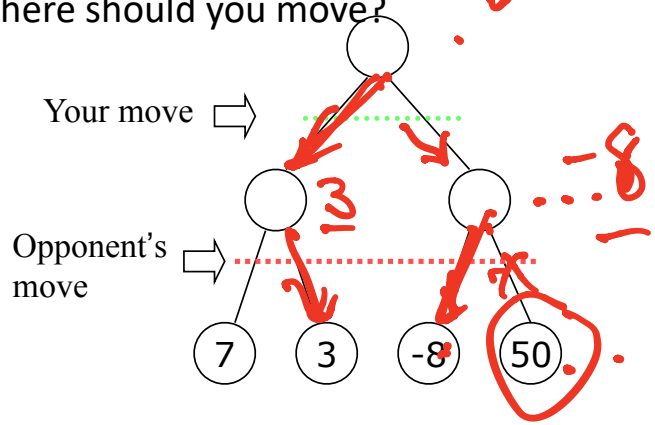
From our opponents point of view, positive numbers are bad, negative numbers are good

A trivial “game”

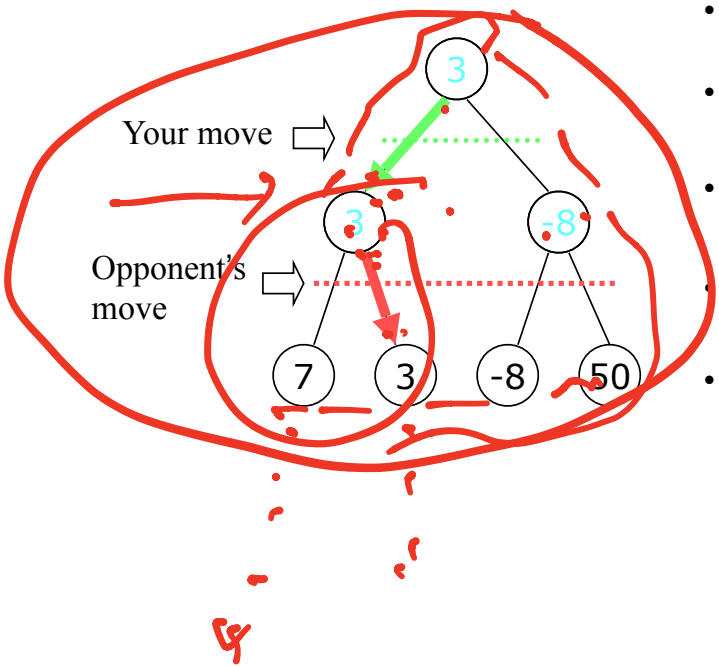
Wouldn’t you like to win 50?

Do you think you will?

Where should you move?



Minimaxing



- Your opponent will choose smaller numbers
- If you move left, your opponent will choose 3
- If you move right, your opponent will choose -8
- Therefore, your choices are really 3 or -8
- You should move left, and play will be as shown

Theory

Consider games with no uncertainty, zero-sum, where players alternate moves

Zermelo (1913): All such games have a deterministic outcome

The optimal strategies are deterministic, and can be obtained through backward induction from the end states

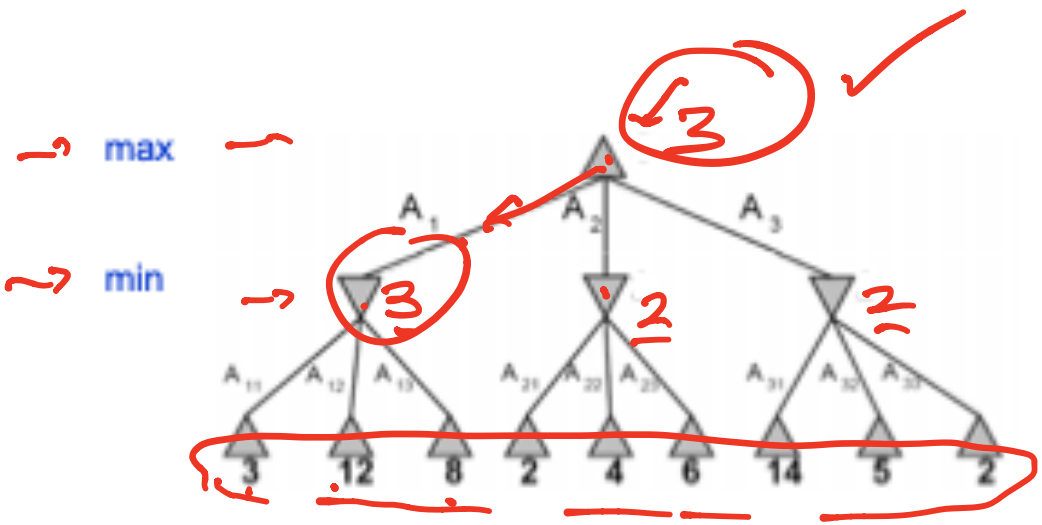
- The minimax algorithm
- Precedes Bellman's principle of optimality by four decades
- Similar in spirit to dynamic programming: solve larger games by extending solutions to smaller games

Minimax Algorithm

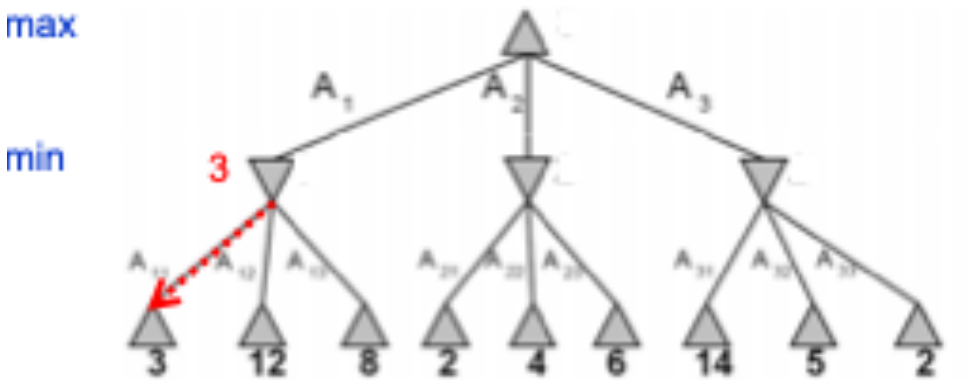
- At each possible "state" in the game, a player is trying to minimize or maximize the final outcome
- Represent the possible game evolutions as a directed game graph of positions, moves.
- The end points of this graph are final states, with a value
- Associated with a state: which player moves at that state
- Min-max algorithm: Let V(s) denote the value of the game at a state, A(s) actions possible at this state, T(s,a) the next state if action a is taken. Then,

$$V(s) = \begin{cases} \min_{a \in A(s)} V(T(s,a)) & \text{if } s \text{ is min player state} \\ \max_{a \in A(s)} V(T(s,a)) & \text{if } s \text{ is max player state} \end{cases}$$

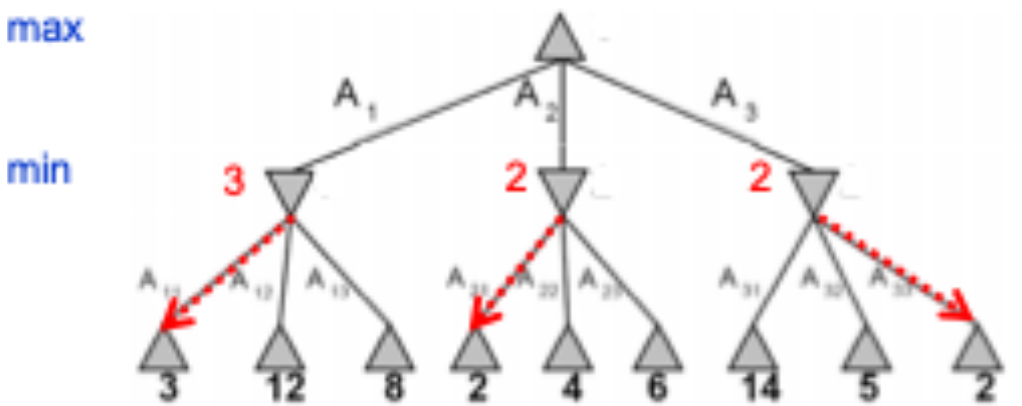
Minimax Example



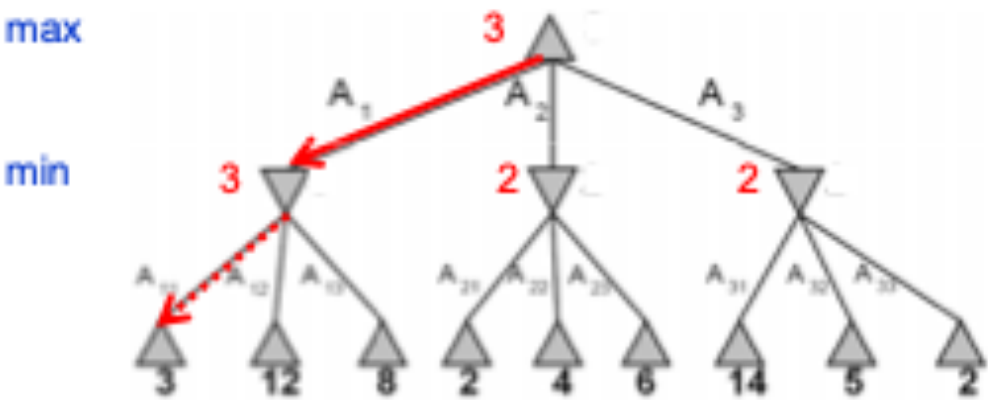
Example - 2



Example-3

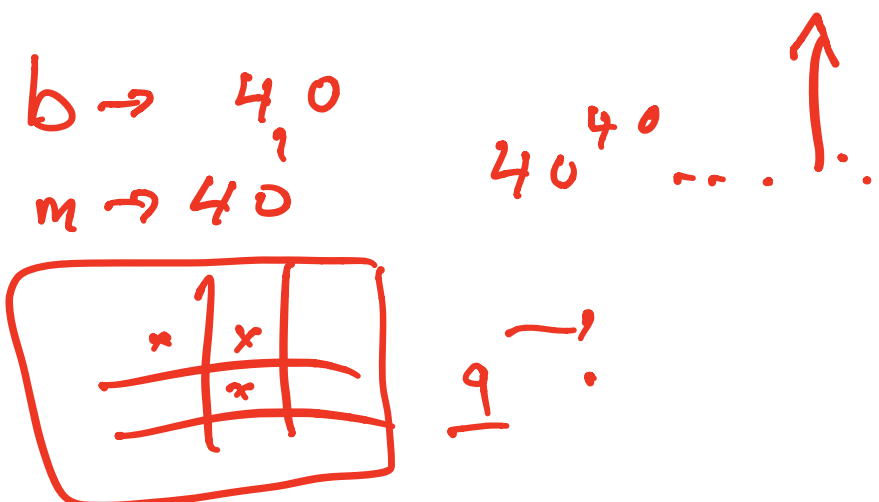


Example-4



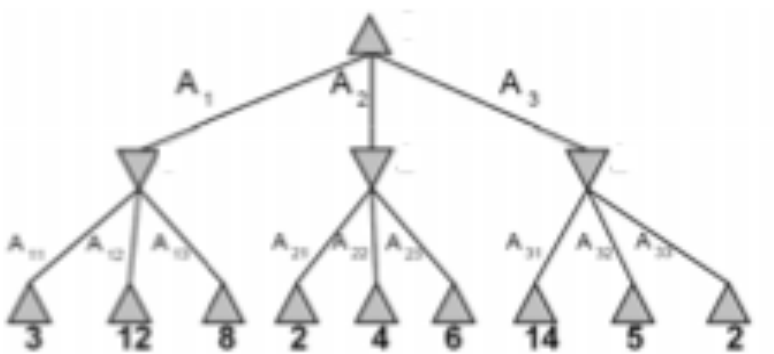
Minimax Properties

- Optimal algorithm under the assumption of zero-sum, perfect information, alternating moves
- Complexity: Size of tree = $O(b^m)$ for total of b moves, m choices per move.
- Algorithm looks at every edge in game tree…
- For tic-tac-toe, b = 9, m = 4 (approximately, using symmetry) → No problem
- For chess, b = 40, m = 100 (approximately) → No way!

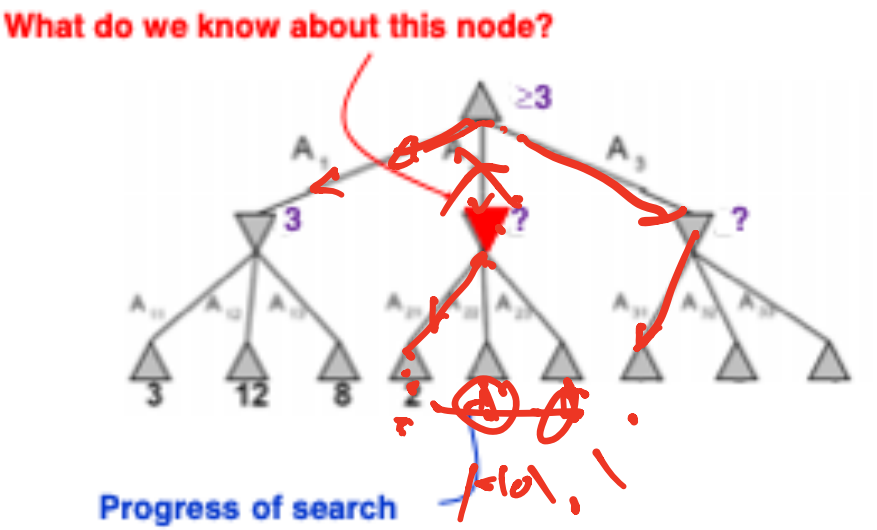


Must we examine the full game tree?

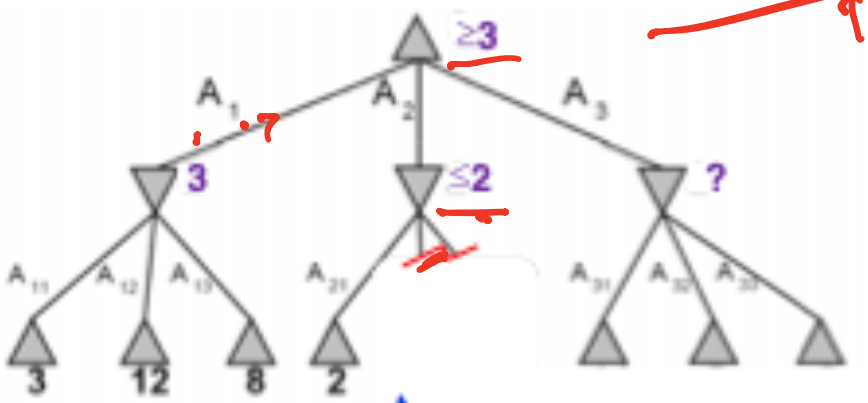
Not really. Do Alpha-Beta pruning- → B+B and



Example


What do we know about this node?
≥3
A1
A2
A3
3
?
?
3
12
8
2
Progress of search

Example-2

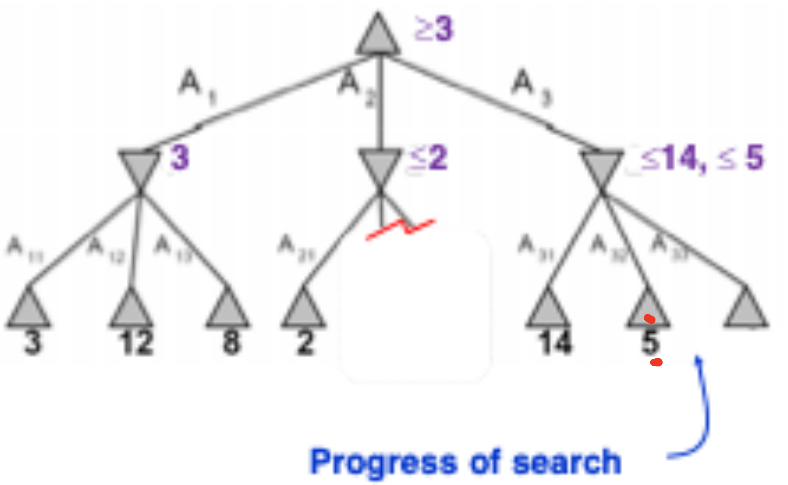


Progress of search

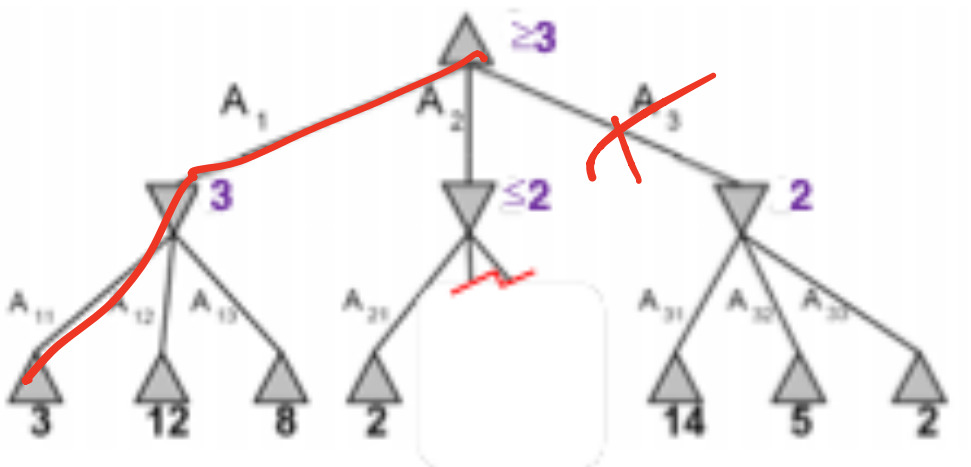
Don't need to look at the other children of middle node: This node will never be reached!!!!

Example-3

Example-4


≥3
A1
A2
A3
3
≤2
≤14, ≤ 5
A11
A12
A13
A21
A31
A32
A33
3
12
8
2
14
5
Progress of search

Example-5



α–β pruning – general case

- Each node has α,β bounds
- **α is the best value found so far that the maximizer player can achieve at paths below that node**
- **β is the smallest value found so far that the minimizer can achieve at paths below that node**
- When exploring possible nodes to move from, the maximizer will not consider nodes where the best value is less than a known value it can achieve by another move
- The minimizer will not consider nodes where the worst value is greater than the worst value it knows it can achieve by another node

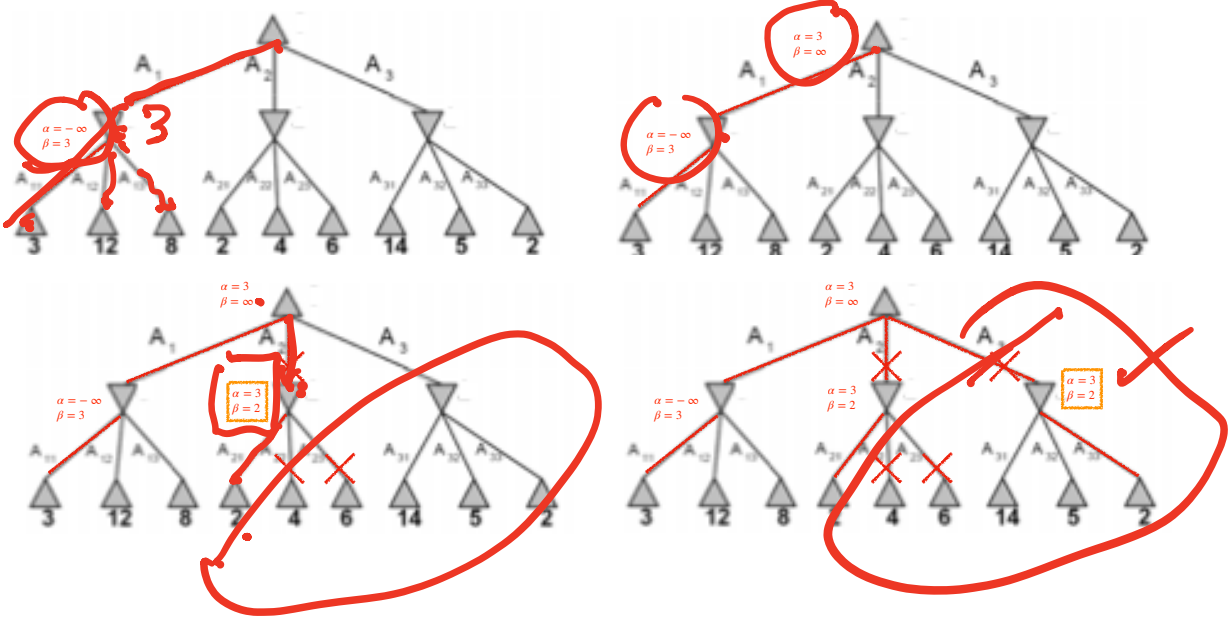
- function minimax(node, depth, isMaximizingPlayer, alpha, beta):

```
if node is a leaf node :
    return value of the node
if isMaximizingPlayer :
    bestVal = -INFINITY
    for each child node :
        value = minimax(node, depth+1, false,
            alpha, beta)
        bestVal = max(bestVal, value)
        alpha = max(alpha, bestVal)
        if beta <= alpha:
            break
    return bestVal
```

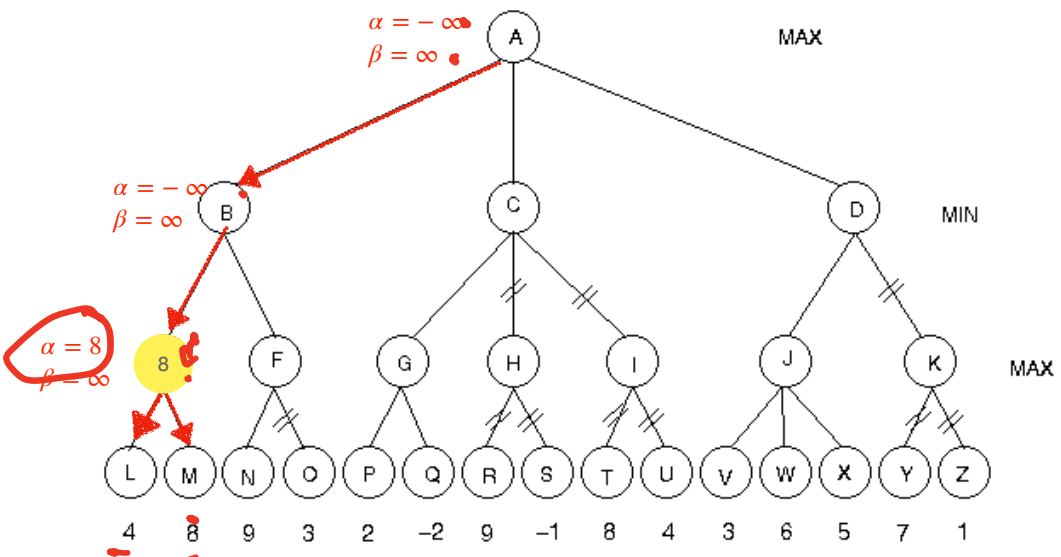
```
else :
    bestVal = +INFINITY
    for each child node :
        value = minimax(node, depth+1,
            true, alpha, beta)
        bestVal = min( bestVal, value)
        beta = min( beta, bestVal)
        if beta <= alpha:
            break
    return bestVal
```

Example: $\alpha - \beta$ Pruning

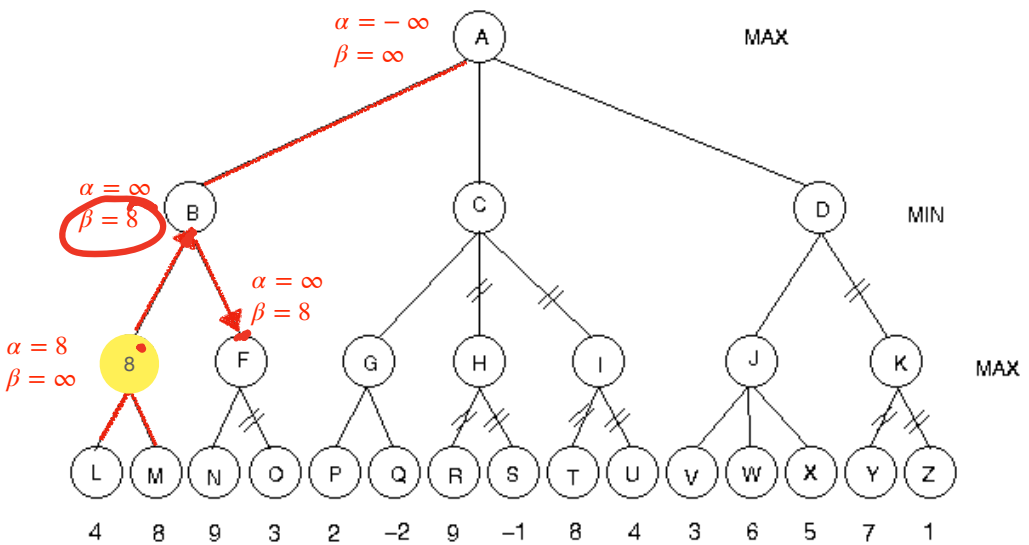
- α is MAX player's best value found so far;
  β is MIN player's best value so far



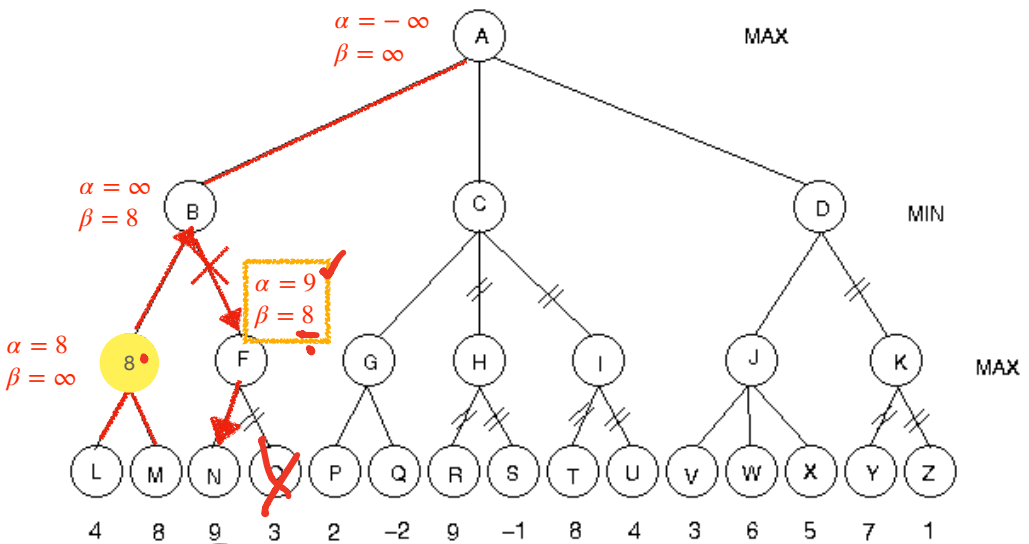
Example


$\alpha = -\infty$
$\beta = \infty$
A
MAX
$\alpha = -\infty$
$\beta = \infty$
B
C
D
MIN
$\alpha = 8$
$\beta = \infty$
8
F
G
H
I
J
K
MAX
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
4
8
9
3
2
−2
9
−1
8
4
3
6
5
7
1

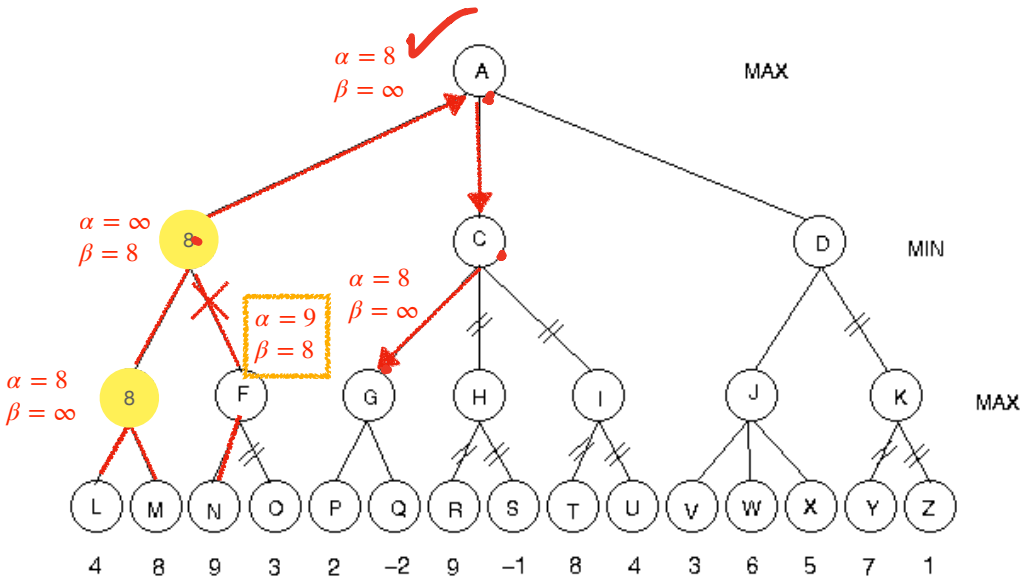
Example


$\alpha = -\infty$
$\beta = \infty$
A
MAX
$\alpha = \infty$
$\beta = 8$
B
C
D
MIN
$\alpha = \infty$
$\beta = 8$
$\alpha = 8$
$\beta = \infty$
8
F
G
H
I
J
K
MAX
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
4 8 9 3 2 −2 9 −1 8 4 3 6 5 7 1

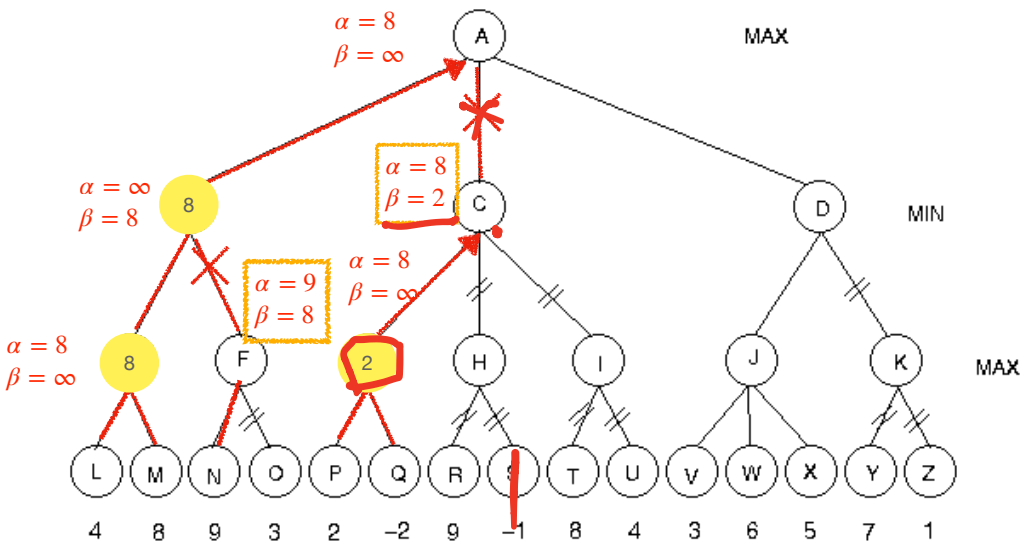
Example


$\alpha = -\infty$
$\beta = \infty$
A
MAX
$\alpha = \infty$
$\beta = 8$
B
C
D
MIN
$\alpha = 9$
$\beta = 8$
$\alpha = 8$
$\beta = \infty$
8
F
G
H
I
J
K
MAX
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
4
8
9
3
2
−2
9
−1
8
4
3
6
5
7
1

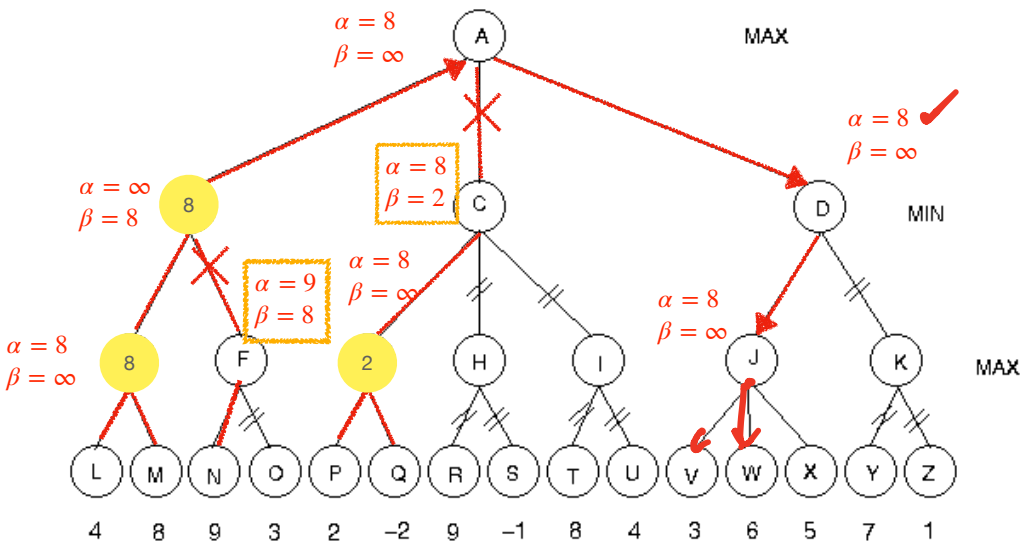
Example


α = 8
β = ∞
A
MAX
α = ∞
β = 8
8
C
D
MIN
α = 8
β = ∞
α = 9
β = 8
α = 8
β = ∞
8
F
G
H
I
J
K
MAX
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
4
8
9
3
2
−2
9
−1
8
4
3
6
5
7
1

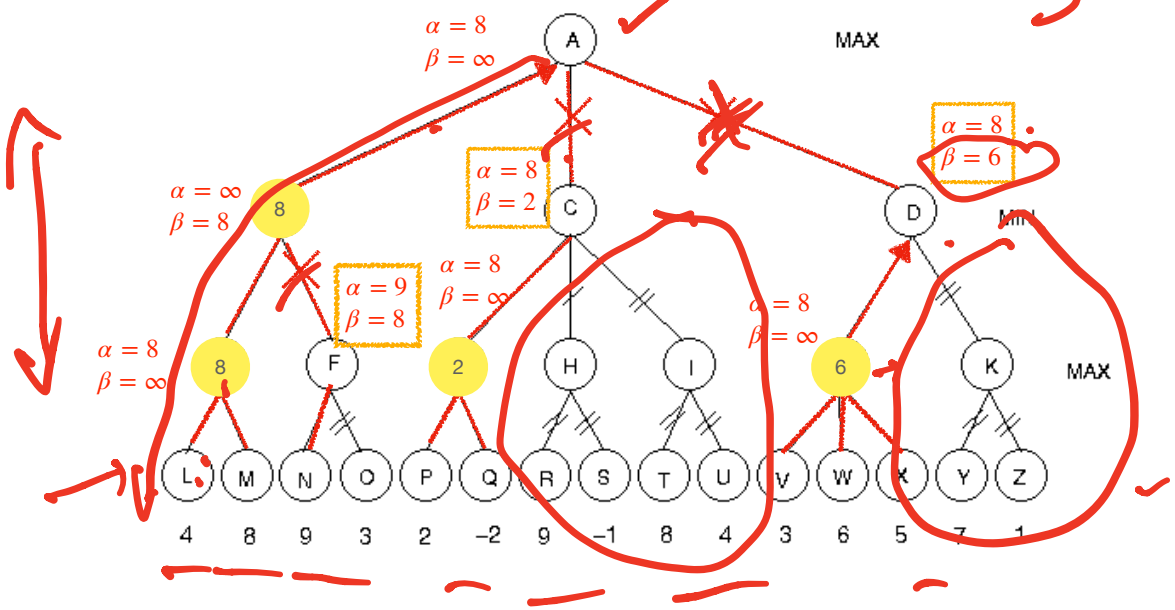
Example


α = 8
β = ∞
A
MAX
α = ∞
β = 8
8
α = 8
β = 2
C
D
MIN
α = 9
β = 8
α = 8
β = ∞
α = 8
β = ∞
8
F
2
H
I
J
K
MAX
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
4
8
9
3
2
−2
9
−1
8
4
3
6
5
7
1

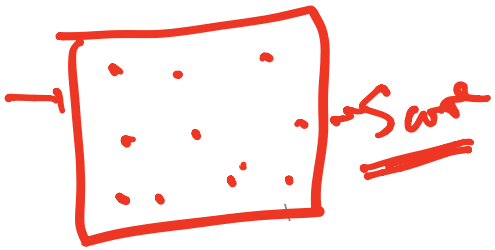
Example


α = 8
β = ∞
A
MAX
α = 8 ✓
β = ∞
α = ∞
β = 8
8
α = 8
β = 2
C
D
MIN
α = 9
β = 8
α = 8
β = ∞
α = 8
β = ∞
α = 8
β = ∞
8
F
2
H
I
J
K
MAX
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
4
8
9
3
2
−2
9
−1
8
4
3
6
5
7
1

Example


A
MAX
B
C
D
MIN
E
F
G
H
I
J
K
MAX
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
4 8 9 3 2 −2 9 −1 8 4 3 6 5 7 1
α = 8
β = ∞
α = ∞
β = 8
α = 8
β = 2
α = 8
β = 6
α = 9
β = 8
α = 8
β = ∞
α = 8
β = ∞
α = 8
β = ∞

Heuristic Value Function



- In a large game, you don't really know the payoffs

  Can't search down to the leaves

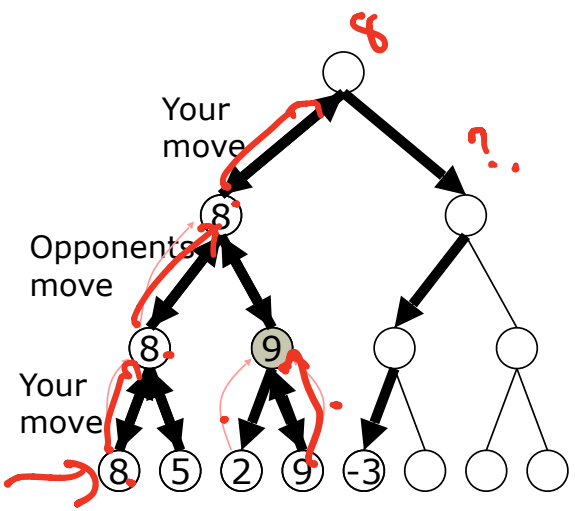
- A heuristic value function computes, for a given node, your best guess as to what the payoff will be
- The heuristic value function uses whatever knowledge you can build into the program
  - Can be used to select which child node to explore next
- We make two key assumptions:
  - Your opponent uses the same heuristic value function as you do
  - The more moves ahead you look, the better your heuristic value function will work

PBVs

A PBV is a preliminary backed-up value

- Explore down to a given level using depth-first search
- As you reach each lowest-level node, evaluate it using your heuristic function
- Back up values to the next higher node, according to the following rules:
  - If it's your move, bring up the largest value, possibly replacing a smaller value
  - If it's your opponent's move, bring up the smallest value, possible replacing a larger value

Using PBVs (animated)

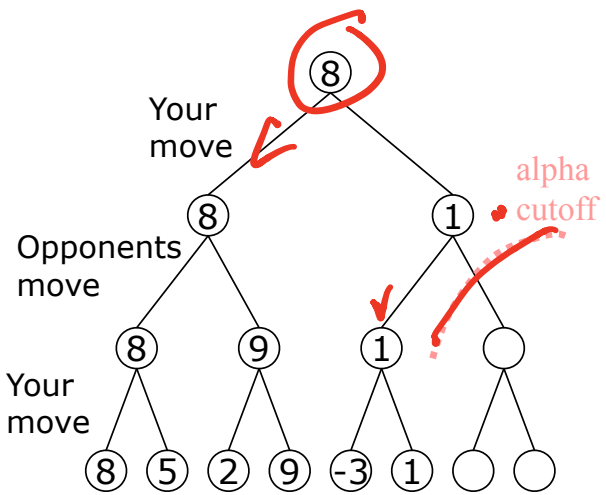


- Do a DFS; find an 8 and bring it up
- Explore 5; smaller than 8, so ignore it
- Backtrack; bring 8 up another level
- Explore 2; bring it up
- Explore 9; better than 2, so bring it up, replacing 2
- 9 is not better than 8 (for your opponent), so ignore it
- Explore –3, bring it up
- Etc.

Bringing up values

- If it's your move, and the next child of this node has a larger value than this node, replace this value
- If it's your opponent's move, and the next child of this node has a smaller value than this node, replace this value
- At your move, never reduce a value
- At your opponent's move, never increase a value

Alpha cutoffs



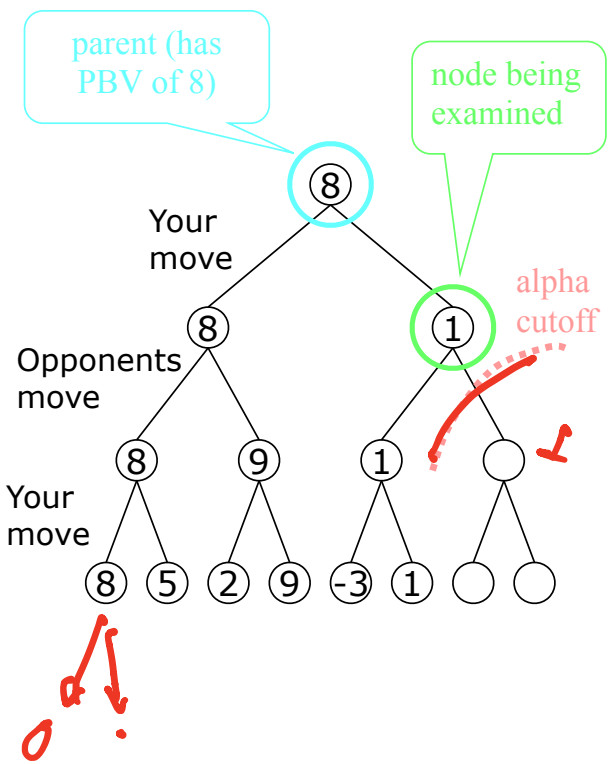
The value at your move is 8 (so far)

If you move right, the value there is 1 (so far)

Your opponent will *never increase* the value at this node; it will always be less than 8

You can *ignore* the remaining nodes

Alpha cutoffs, in more detail



You have an alpha cutoff when:

- You are examining a node at which it is your opponent's move, *and*
- You have a PBV for the node's parent, *and*
- You have brought up a PBV that is less than the PBV of the node's parent, *and*
- The node has other children (which we can now "prune")

Beta cutoffs

An alpha cutoff occurs where

- It is *your opponent's turn* to move
- You have computed a PBV for this node's parent
- The node's parent has a *higher PBV* than this node
- This node has other children you haven't yet considered

A beta cutoff occurs where

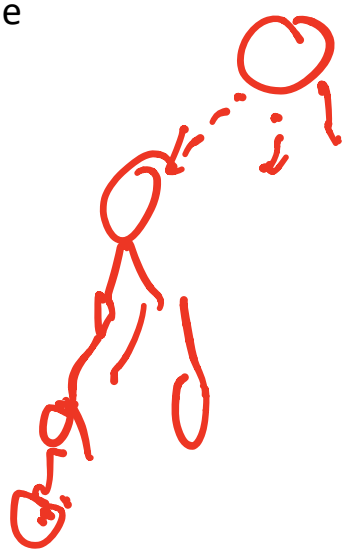
- It is *your turn* to move
- You have computed a PBV for this node's parent
- The node's parent has a *lower PBV* than this node
- This node has other children you haven't yet considered

Using beta cutoffs

- Beta cutoffs are harder to understand, because you have to see things from your opponent's point of view
- Your opponent's alpha cutoff is your beta cutoff
- We assume your opponent is rational, and is using a heuristic function similar to yours
- Even if this assumption is incorrect, it's still the best we can do

The importance of cutoffs

- If you can search to the end of the game, you know exactly how to play
- The further ahead you can search, the better
- If you can *prune* (ignore) large parts of the tree, you can search deeper on the other parts
- Since the number of nodes at each level grows exponentially, the higher you can prune, the better
- You can save exponential time



Heuristic alpha-beta searching

The higher in the search tree you can find a cutoff, the better (because of exponential growth)

To maximize the number of cutoffs you can make:

- Apply the heuristic function at each node you come to, not just at the lowest level
- Explore the “best” moves first
- “Best” means best for the player *whose move it is at that node*

Best game playing strategies

- For any game much more complicated than tic-tac-toe, you have a time limit
- Searching takes time; you need to use heuristics to minimize the number of nodes you search
- But complex heuristics take time, reducing the number of nodes you can search
- Seek a balance between simple (but fast) heuristics, and slow (but good) heuristics

Heuristic for pacman?

What features would be good for Pacman?

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Other games…

- N-player games – monopoly, …
- Non-zero sum games – bargaining, etc.
- Stochastic games – some moves by random elements
  - Card games
  - Backgammon, roulette
- Games with imperfect information
  - Stratego, battleship, bridge, poker
- Hard to find recursive solution techniques like minimax, so can't use pruning
  - More computation