# Hill-Climbing

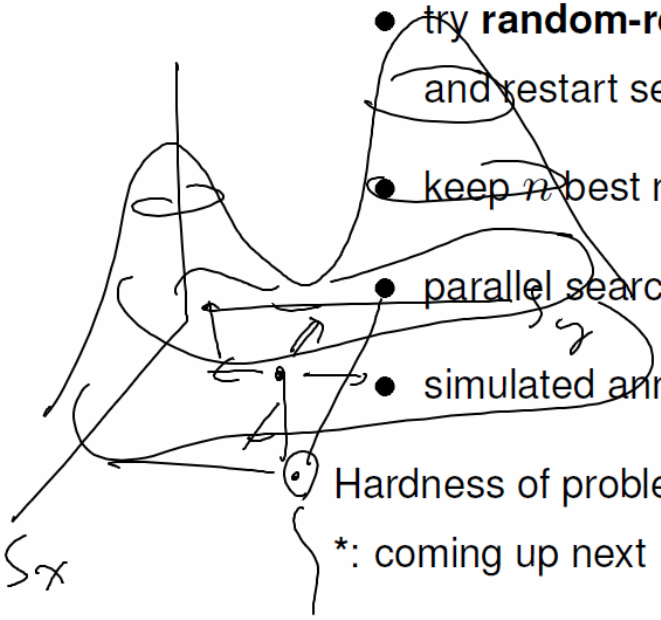
- no queue, keep only the best node
- greedy, no back-tracking
- good for domains where **all nodes are solutions**:
  - goal is to **improve** quality of the solution
  - optimization problems
- note that it is different from greedy best-first search that uses the heuristic function, which keeps a node list

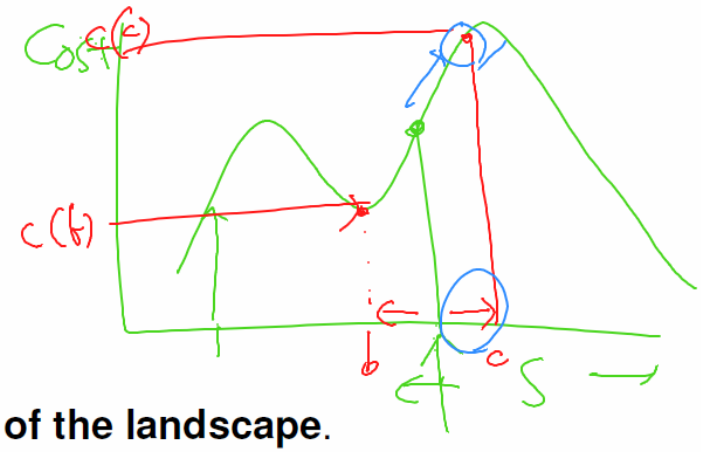
These nodes

# Hill-Climbing Strategies

Problems of local maxima, plateau, and ridges:

- try **random-restart**: move to a random location in the landscape and restart search from there
- keep $n$ best nodes (beam search)
- parallel search
- simulated annealing *



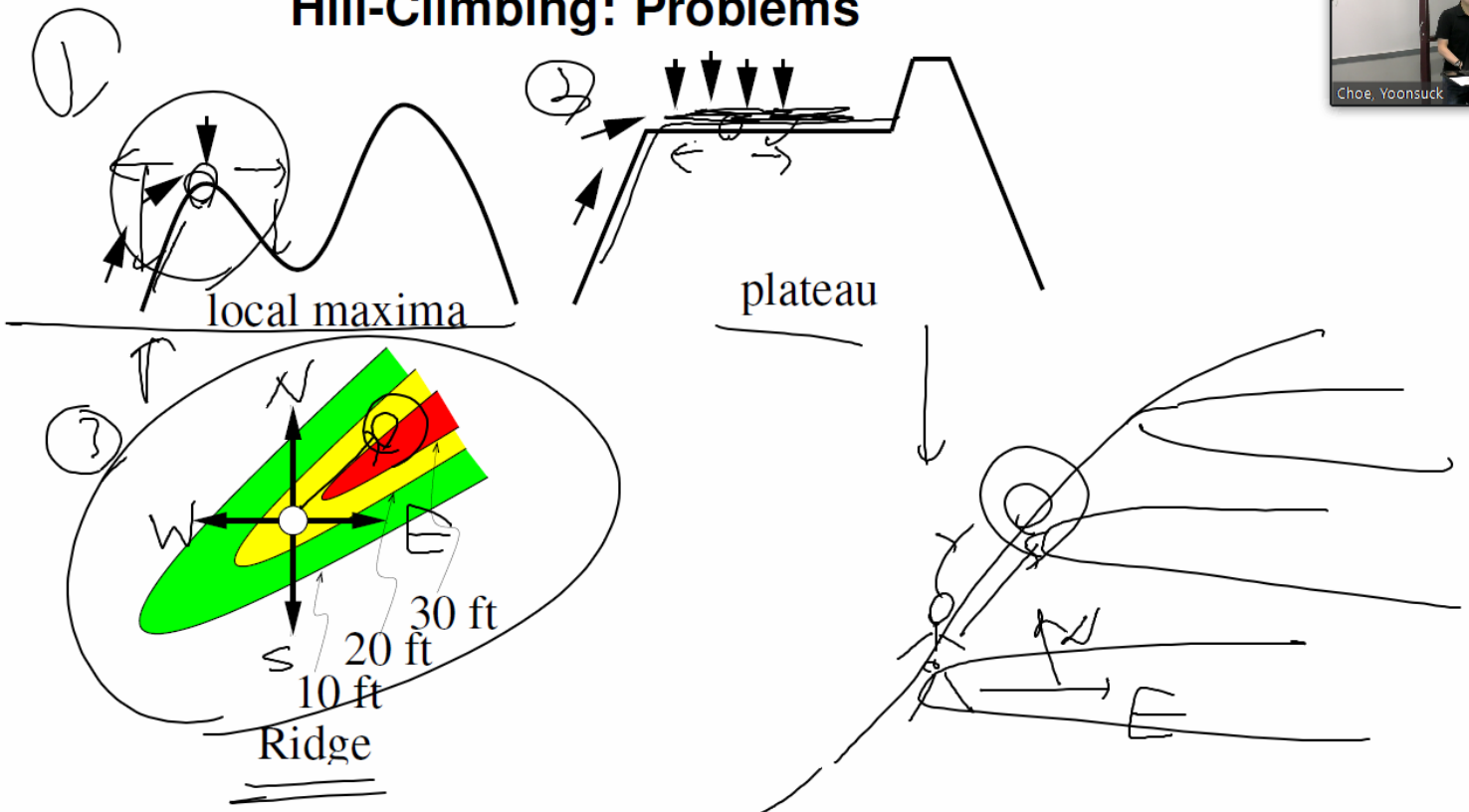



Hardness of problem depends on the **shape of the landscape**.

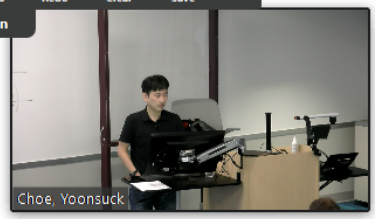
*: coming up next

$a = (a_x, a_y)$

# Hill-Climbing: Problems



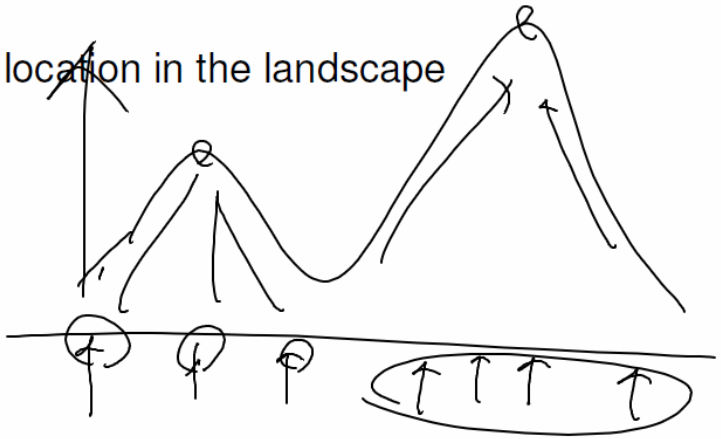

- Possible solution: **simulated annealing** – gradually decrease randomness of move to attain globally optimal solution (more on this next week).




# Hill-Climbing Strategies

Problems of local maxima, plateau, and ridges:

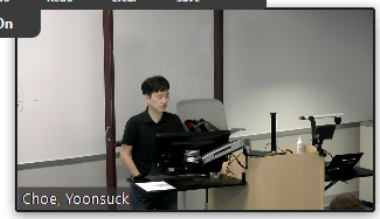
- try **random-restart**: move to a random location in the landscape and restart search from there
- keep $n$ best nodes (beam search)
- parallel search
- simulated annealing *



Hardness of problem depends on the **shape of the landscape**.

*: coming up next

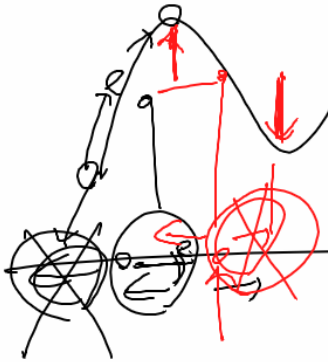
Then you can do this paddle as such and the final solution is it's called simulated the kneeling




# Simulated Annealing: Overview

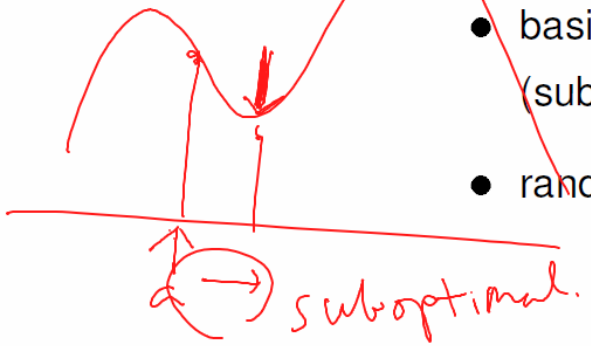
Annealing:

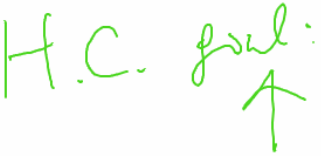
- heating metal to a high-temperature (making it a liquid) and then allowing to cool slowly (into a solid); this relieves internal stresses and results in a more stable, lower-energy state in the solid.
- at high temperature, atoms move actively (large distances with greater randomness), but as temperature is lowered, they become more static.

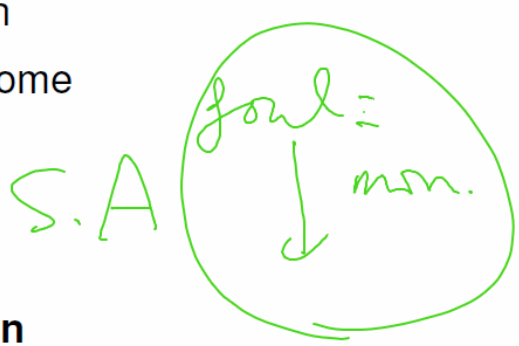


Simulated annealing is similar:

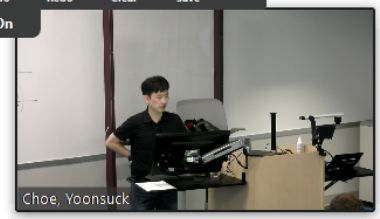
- basically, hill-climbing with randomness that allows going **down** (suboptimal) as well as the desired **up** (optimal)
- randomness (as temperature) is reduced over time





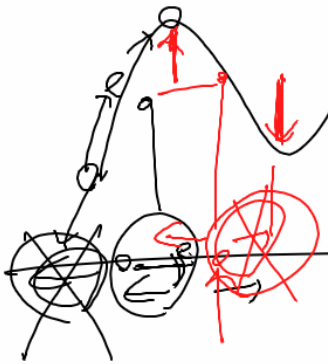
As I mentioned, you can flip it either way, and does that the same thing

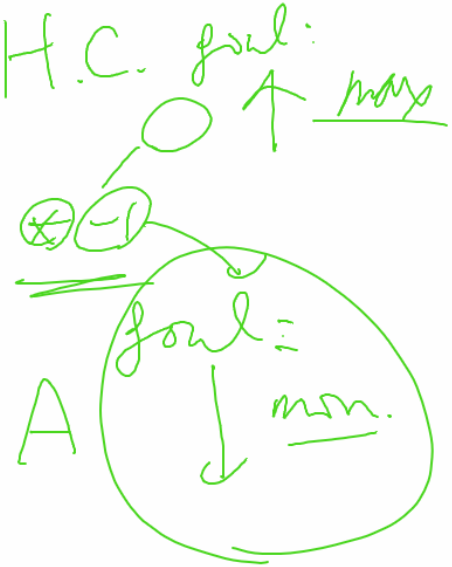



# Simulated Annealing: Overview

Annealing:

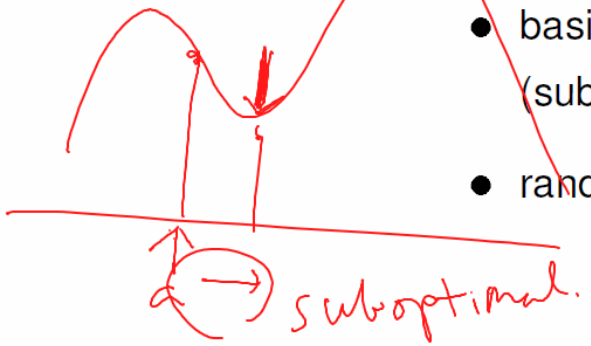
- heating metal to a high-temperature (making it a liquid) and then allowing to cool slowly (into a solid); this relieves internal stresses and results in a more stable, lower-energy state in the solid.
- at high temperature, atoms move actively (large distances with greater randomness), but as temperature is lowered, they become more static.



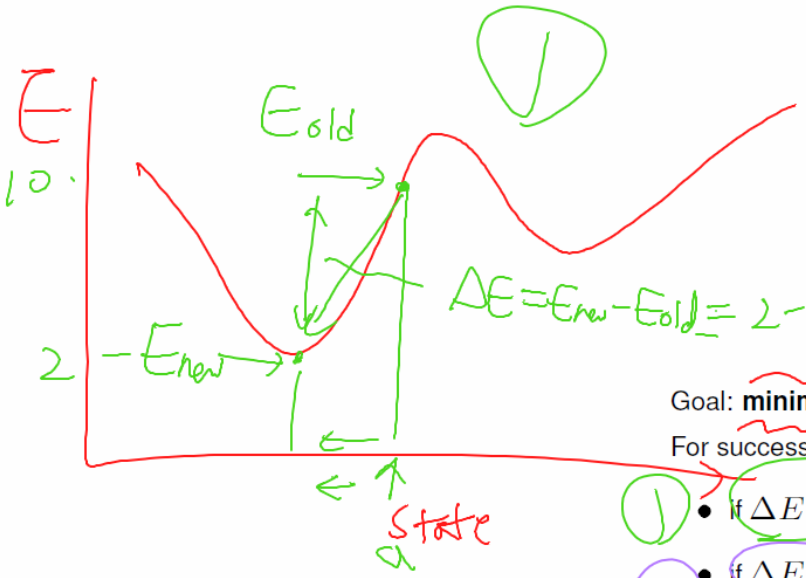


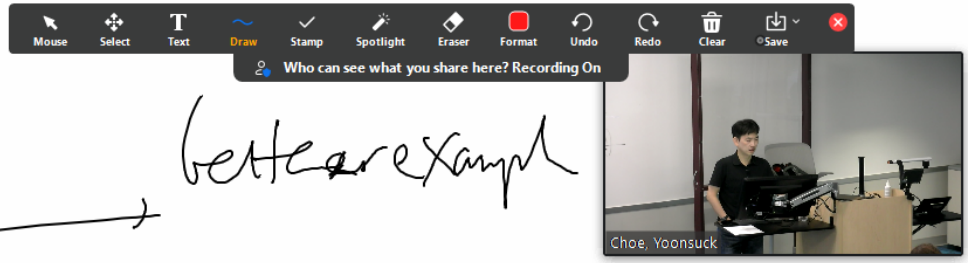

Simulated annealing is similar:

- basically, hill-climbing with randomness that allows going **down** (suboptimal) as well as the desired **up** (optimal)
- randomness (as temperature) is reduced over time




You can make it into this minimization goal. Niceization can be turned into minimization by just multiplying the objective function or the cost is




better example

## Simulated Annealing (SA)

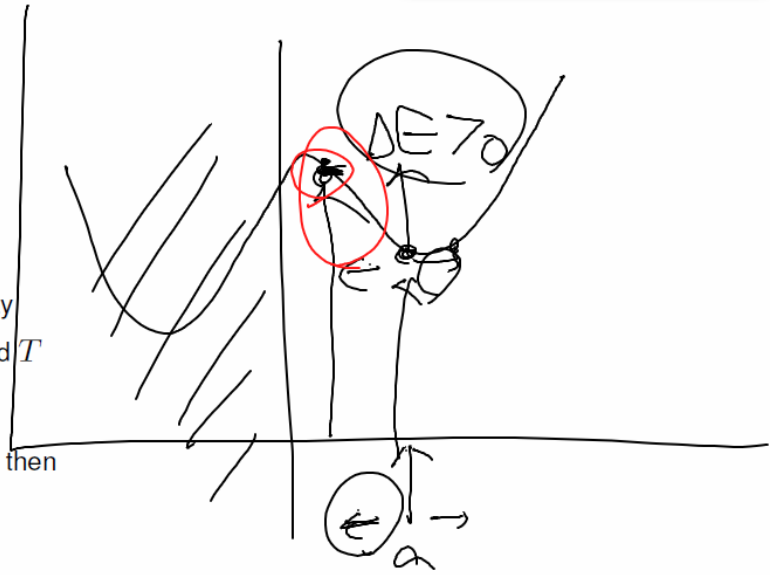
Goal: **minimize** the energy $E$, as in statistical thermodynamics.

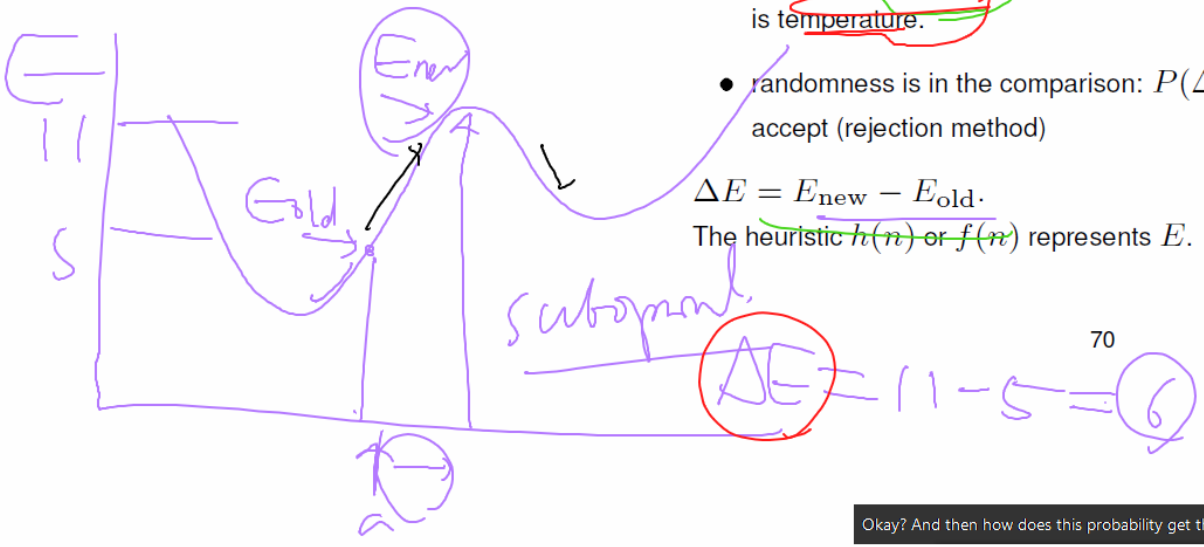
For successors of the current node,

1. if $\Delta E \leq 0$ (desired), the move is accepted
2. if $\Delta E > 0$ (undesired), the move is accepted with probability $P(\Delta E) = e^{-\frac{\Delta E}{kT}}$, where $k$ is the Boltzmann constant and $T$ is temperature.
   - randomness is in the comparison: $P(\Delta E) < \text{rand}(0, 1)$, then accept (rejection method)

$\Delta E = E_{\text{new}} - E_{\text{old}}$.

The heuristic ~~$h(n)$ or $f(n)$~~ represents $E$.






Okay? And then how does this probability get the term

You are screen sharing
Stop Share

$\Delta E > 0$

## Temperature and $P(\Delta E) < \text{rand}(0, 1)$

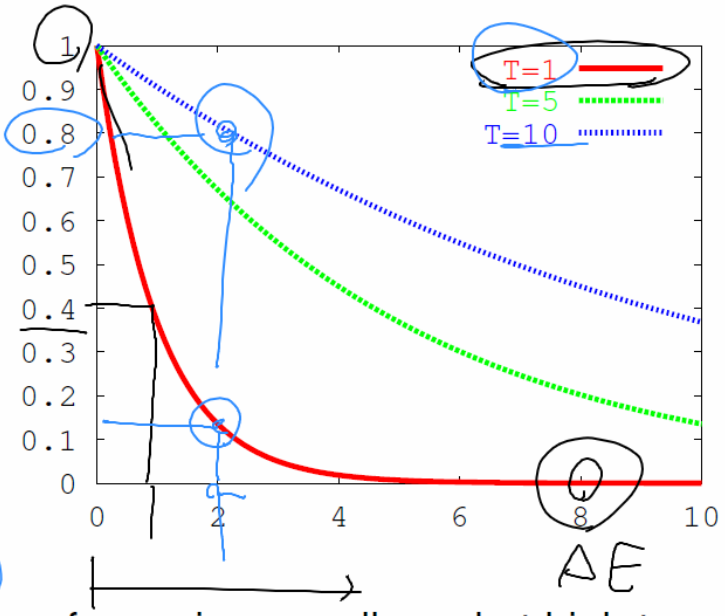
1 a $\Delta E = 8$ ≈ 0.0

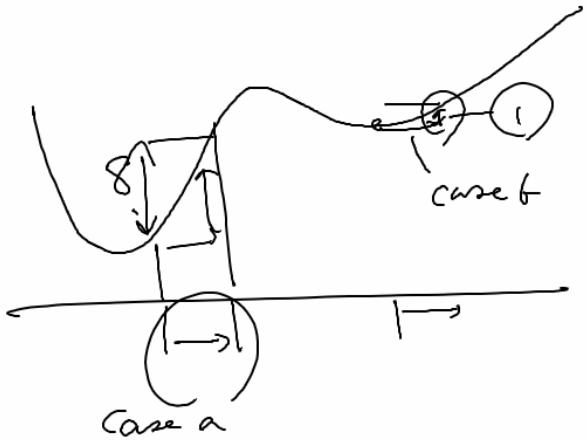
b $\Delta E = 1$ ≈ 0.4

2 $\Delta E = 2$.

a $T = 1$ $p \approx 0.15$

b $T = 10$ $p \approx 0.8$

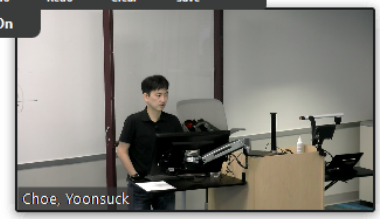


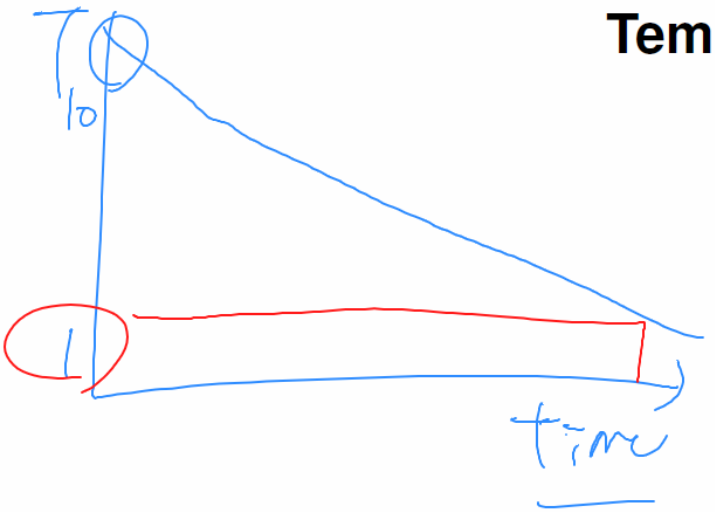

Downward moves of any size are allowed at high temperature, but at low temperature, only small downward moves are allowed.

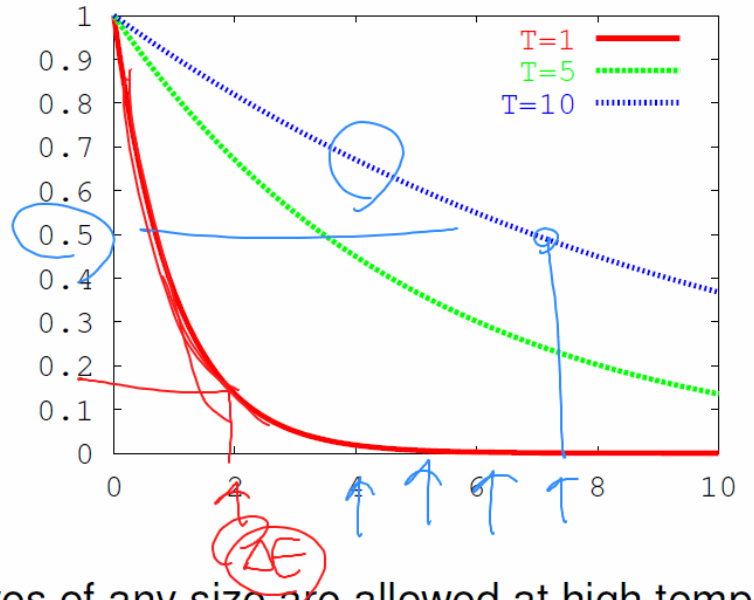
- Higher temperature $T \rightarrow$ higher probability of suboptimal move
- Lower $\Delta E \rightarrow$ higher probability of suboptimal move

chance of accepting that move which is suboptimal with the Delta e is very low, then there's a higher chance of accept accepting the sub optimal move



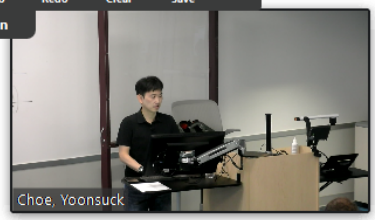

# Temperature and $P(\Delta E) < \text{rand}(0, 1)$



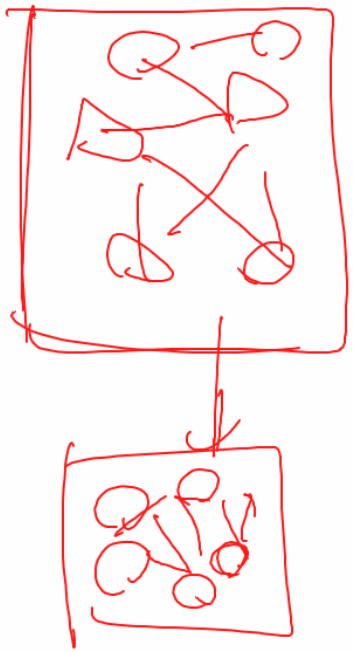




Downward moves of any size are allowed at high temperature, but at low temperature, only small downward moves are allowed.

- Higher temperature $T \rightarrow$ higher probability of suboptimal move
- Lower $\Delta E \rightarrow$ higher probability of suboptimal move




# Simulated Annealing Applications



- VLSI wire routing and placement
- Various scheduling optimization tasks
- Traffic control
- Neural network training
- etc.

And and also this this kind of concept has been used in training neural networks

# $IDA^*$

**function** $IDA^*$(*problem*)

*root* ← Make-Node(Initial-State(*problem*))

*f-limit* ← f-Cost(*root*)

**loop do**

*solution, f-limit* ← DFS-Contour(*root, f-limit*)

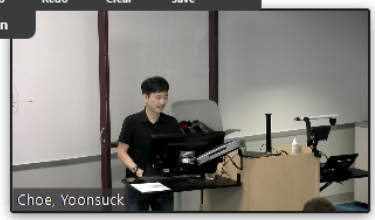
**if** *solution* != NULL **then return** *solution*

**if** *f-limit* == $\infty$ **then return** *failure*

**end loop**

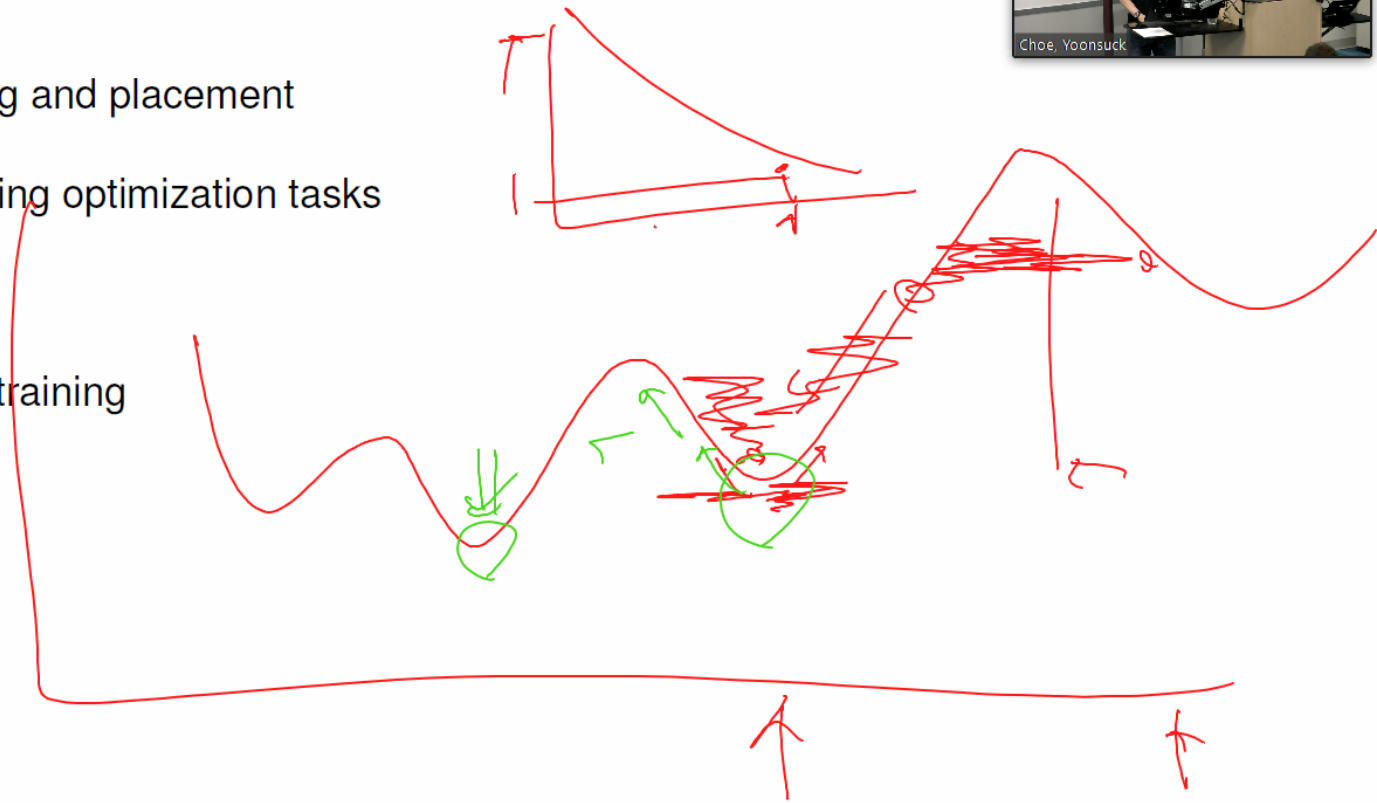
Basically, iterative deepening depth-first-search with depth defined as the $f$-cost ($f = g + n$):

So if you go very, very deep, then basically, this cool spec you to this kind of request to call would be your memory footprint

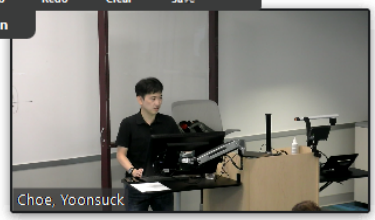



# Simulated Annealing Applications

- VLSI wire routing and placement
- Various scheduling optimization tasks
- Traffic control
- Neural network training
- etc.



Yeah. Well, I guess it depends depends on manufacturing right




# Simulated Annealing (SA)

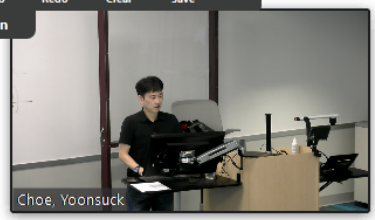
Goal: **minimize** the energy $E$, as in statistical thermodynamics.

For successors of the current node,

- if $\Delta E \leq 0$ (desired), the move is accepted
- if $\Delta E > 0$ (undesired), the move is accepted with probability $P(\Delta E) = e^{-\frac{\Delta E}{kT}}$, where $k$ is the Boltzmann constant and $T$ is temperature.
- randomness is in the comparison: $P(\Delta E) < \text{rand}(0, 1)$, then accept (rejection method)

$\Delta E = E_{\text{new}} - E_{\text{old}}$.

The heuristic $h(n)$ or $f(n)$ represents $E$.




# Simulated Annealing (SA)

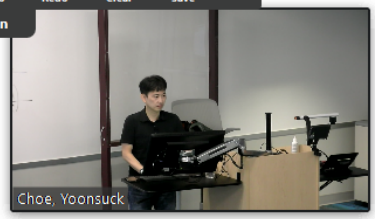
Goal: **minimize** the energy $E$, as in statistical thermodynamics.

For successors of the current node,

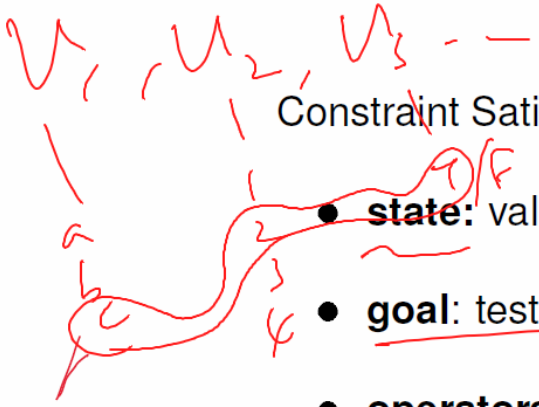
- if $\Delta E \leq 0$ (desired), the move is accepted
- if $\Delta E > 0$ (undesired), the move is accepted with probability $P(\Delta E) = e^{-\frac{\Delta E}{kT}}$, where $k$ is the Boltzmann constant and $T$ is temperature.
- randomness is in the comparison: $P(\Delta E) < \text{rand}(0, 1)$, then accept (rejection method)

$$\Delta E = E_{\text{new}} - E_{\text{old}}.$$

The heuristic $h(n)$ or $f(n)$ represents $E$.



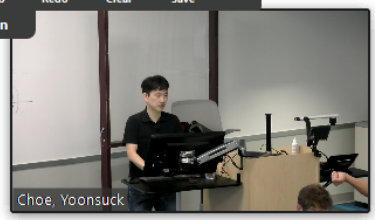

# Constraint Satisfaction Search



Constraint Satisfaction Problem (CSP):

- **state:** values of a set of **variables**
- **goal**: test if a set of constraints are met
- **operators**: set values of variables
- general search can be used, but specialized solvers for CSP work better

So all these constraints are met. The operators are basically the act of setting a particular value to the particular

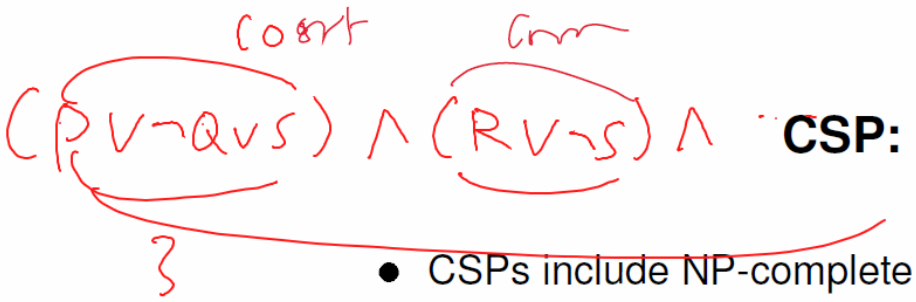


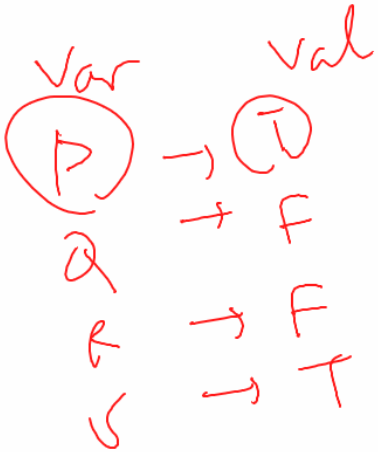

# Constraints

- Unary, binary, and higher order constraints: how many variables should simultaneously meet the constraint
- Absolute constraints vs. preference constraints
- Variables are defined in a certain **domain**, which determines the possible set of values, either discrete or continuous.

This is part of a much more complex problem called **constrained optimization problems** in operations research consisting of cost function (either minimize or maximize) and several constraints. Problems can be linear, nonlinear, convex, nonconvex, etc. Straight-forward solutions exist for a limited subclass of these (for example, for linear programming problems can be solved by the simplex method).

But this is a little bit too detailed



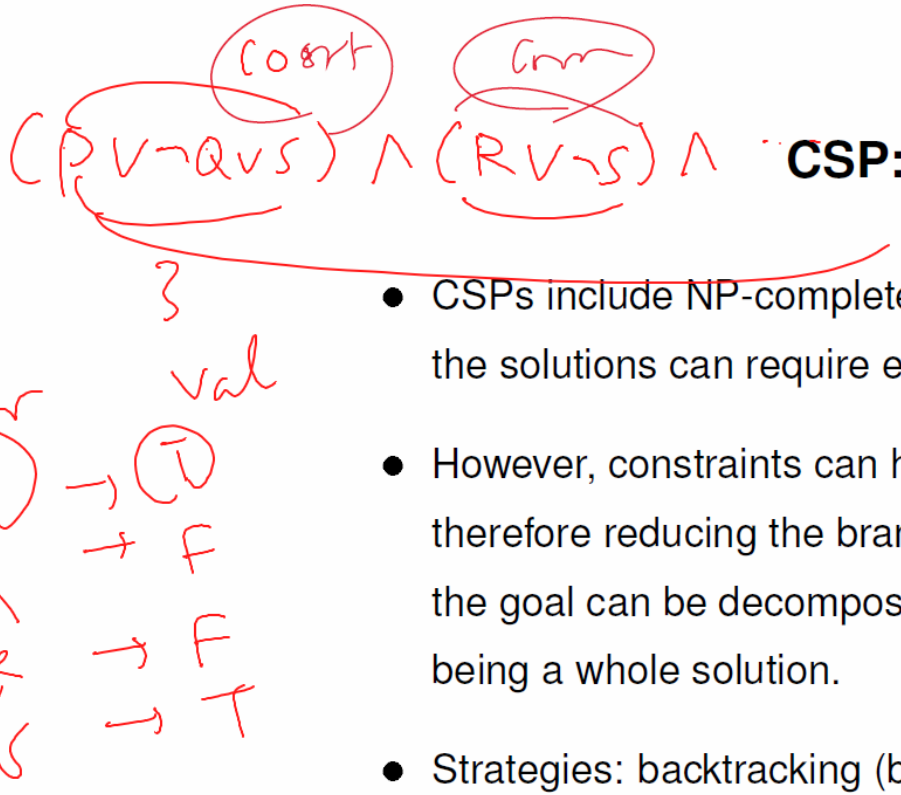




# CSP: continued

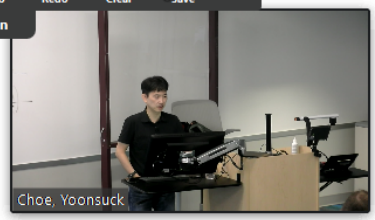
- CSPs include NP-complete problems such as 3-SAT, thus finding the solutions can require exponential time.
- However, constraints can help narrow down the possible options, therefore reducing the branching factor. This is because in CSP, the goal can be decomposed into several constraints, rather than being a whole solution.
- Strategies: backtracking (back up when constraint is violated), forward checking (do not expand further if look-ahead returns a constraint violation). Forward checking is often faster and simple to implement.



So these are the variables. These are the values. And basically these are the constraints



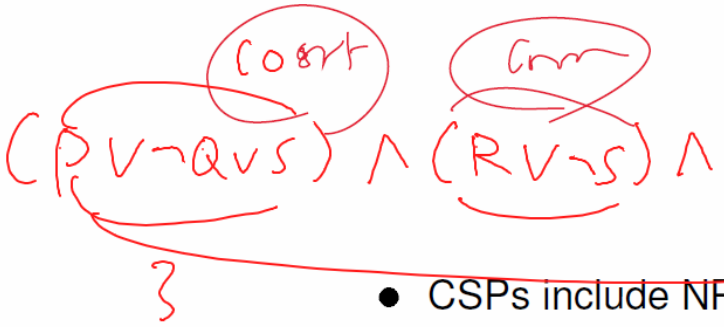


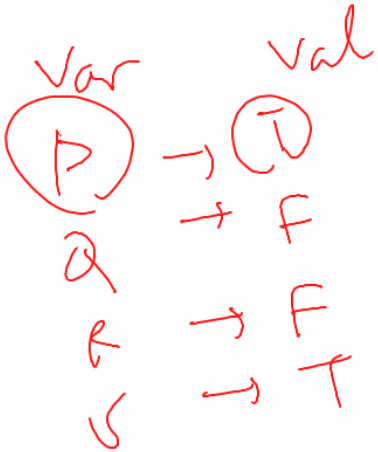




# CSP: continued

- CSPs include NP-complete problems such as 3-SAT, thus finding the solutions can require exponential time.
- However, constraints can help narrow down the possible options, therefore reducing the branching factor. This is because in CSP, the goal can be decomposed into several constraints, rather than being a whole solution.
- Strategies: backtracking (back up when constraint is violated), forward checking (do not expand further if look-ahead returns a constraint violation). Forward checking is often faster and simple to implement.

So in practice, it can be done more




# CSP: continued

- CSPs include NP-complete problems such as 3-SAT, thus finding the solutions can require exponential time.
- However, constraints can help narrow down the possible options, therefore reducing the branching factor. This is because in CSP, the goal can be decomposed into several constraints, rather than being a whole solution.
- Strategies: backtracking (back up when constraint is violated), forward checking (do not expand further if look-ahead returns a constraint violation). Forward checking is often faster and simple to implement.

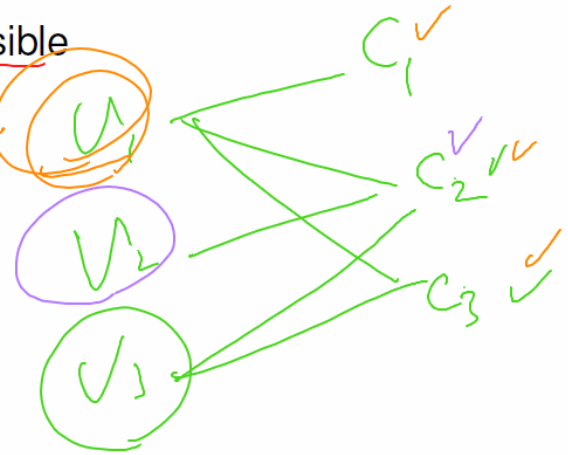


So in practice, it can be done much more efficiently, and there are several strategies like backtracking, or for checking, as you assign these values to the


# Heuristics for Constraint Satisfaction Problems

General strategies for **variable selection**:

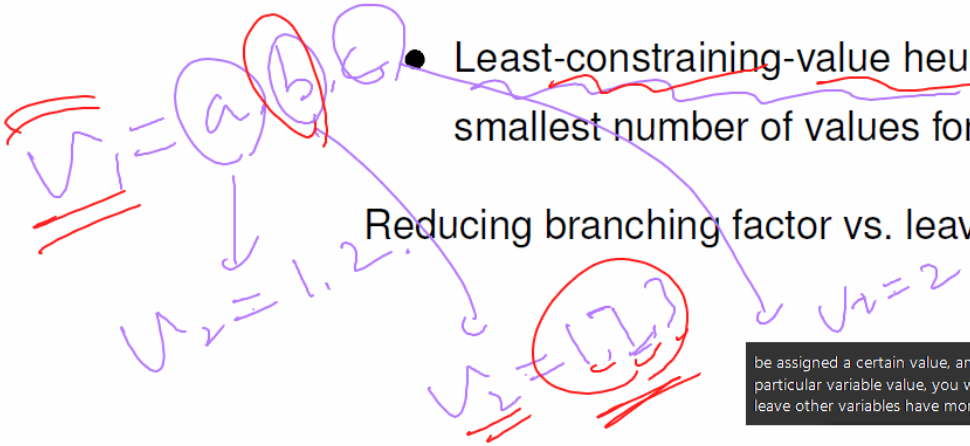
- Most-constrained-variable heuristic (variable with fewest possible values)
- Most-constraining-variable heuristic (variable involved in the largest number of constraints)



and for **value assignment**:

- Least-constraining-value heuristic (value that rules out the smallest number of values for other variables)

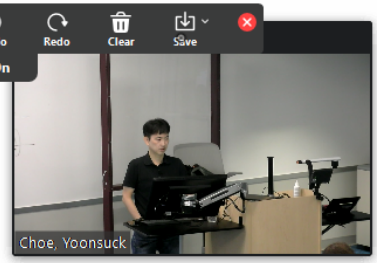
Reducing branching factor vs. leaving freedom for future choices.



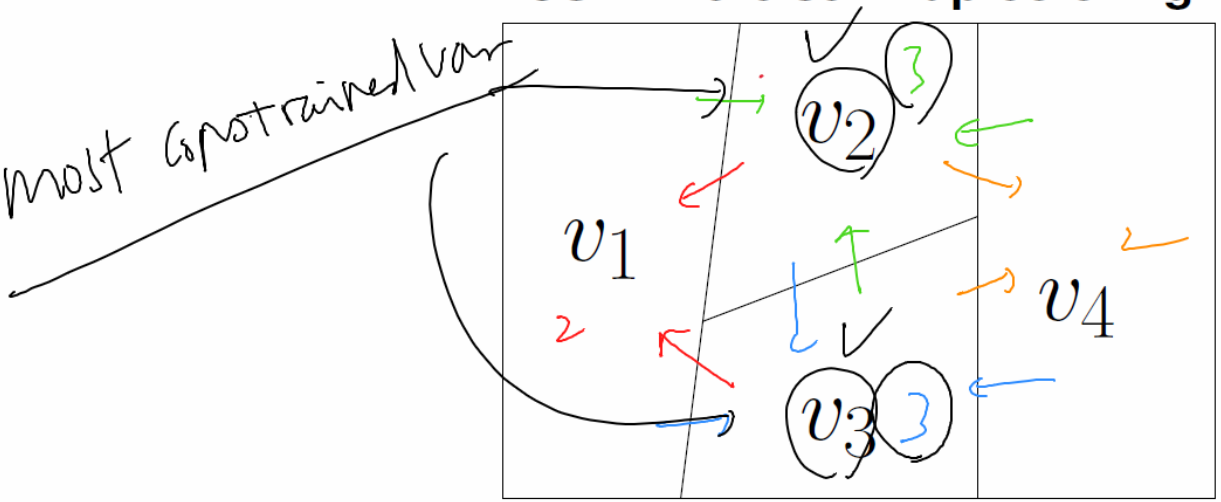
be assigned a certain value, and in trying to pick a particular variable value, you want to pick the one that will leave other variables have more choice

Values = R, G, B

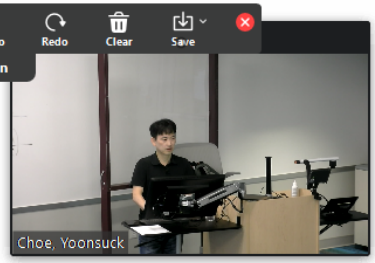





# CSP Exercise: Map coloring




- Variables: $v_1$, $v_2$, $v_3$, and $v_4$.
- Values: Red, Blue, Green
- Constraints: Color map so that no border shares the same color
- Exercises: Variable selection, Value assignment under different conditions.




# DFS-Contour(*root, f-limit*)

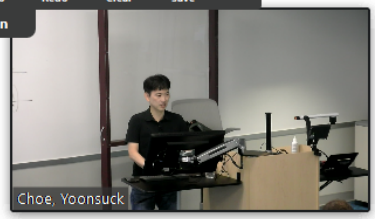
**function** DFS-Contour (**root, f-limit**): returns **solution, f-limit**

Find solution from node **root**, within the $f$-cost limit of **f-limit**. DFS-Contour returns **solution sequence** and **new $f$-cost limit**.

- if $f$-cost(**root**) $>$ **f-limit**, return **null solution** and the $f$-cost(**root**).
- if **root** is a goal node, return solution and its $f$-cost(**root**).
- **recursively call** on all successors, and find successor with minimum $f$-cost. Return its solution and its $f$-cost.

Similar to the recursive implementation of DFS.

That. So that was the Dfs control function which we see that there is this recursive call




# Heuristics for Constraint Satisfaction Problems

General strategies for **variable selection**:

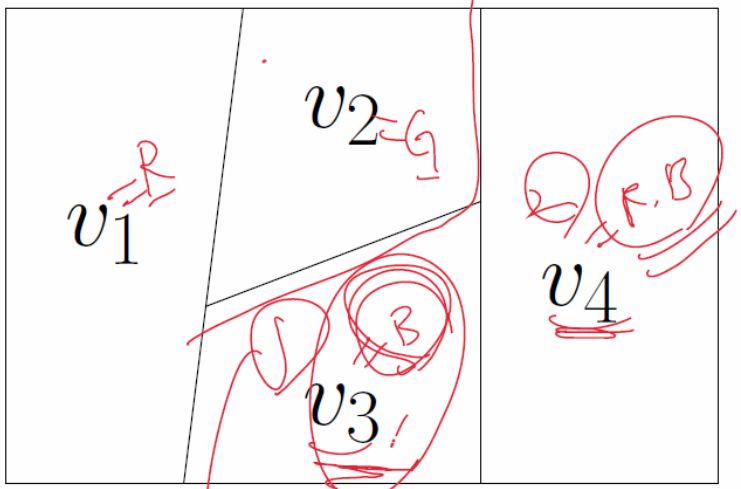
- Most-constrained-variable heuristic (variable with fewest possible values)
- Most-constraining-variable heuristic (variable involved in the largest number of constraints)

and for **value assignment**:

- Least-constraining-value heuristic (value that rules out the smallest number of values for other variables)

Reducing branching factor vs. leaving freedom for future choices.


# CSP Exercise: Map coloring

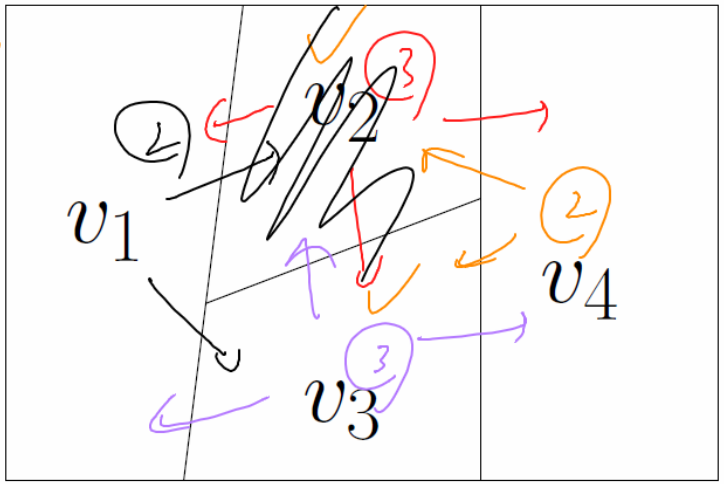


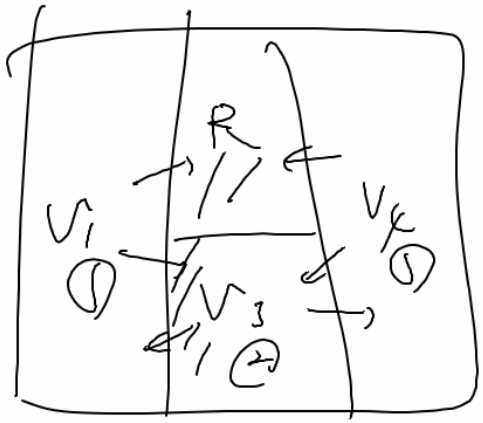

- Variables: $v_1$, $v_2$, $v_3$, and $v_4$.
- Values: Red, Blue, Green
- Constraints: Color map so that no border shares the same color
- Exercises: Variable selection, Value assignment under different conditions.


Most constraining variable

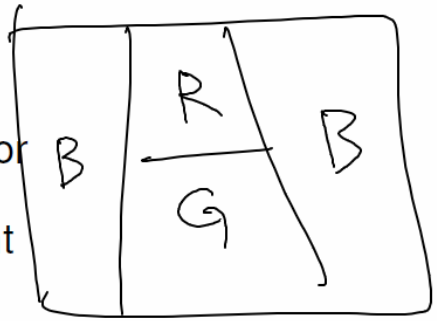
$V_2$ or $V_3$

# CSP Exercise: Map coloring




- Variables: $v_1$, $v_2$, $v_3$, and $v_4$.
- Values: Red, Blue, Green
- Constraints: Color map so that no border shares the same color
- Exercises: Variable selection, Value assignment under different conditions.




It doesn't matter then. Now we can quickly feel that the rest out blue and blue blue


# Heuristics for Constraint Satisfaction Problems

General strategies for **variable selection**:

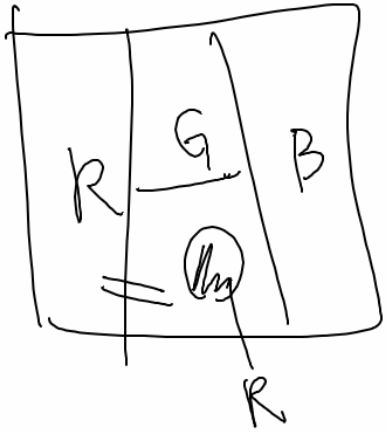
- Most-constrained-variable heuristic (variable with fewest possible values)
- Most-constraining-variable heuristic (variable involved in the largest number of constraints)

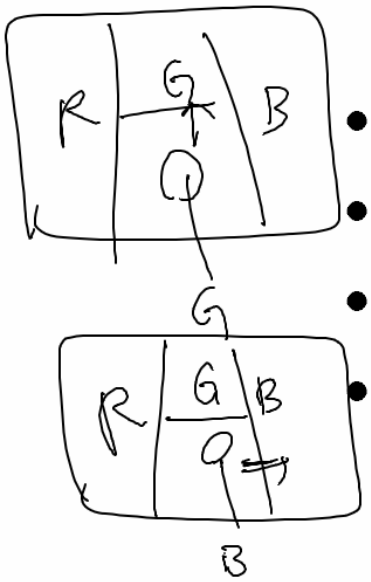
and for **value assignment**:

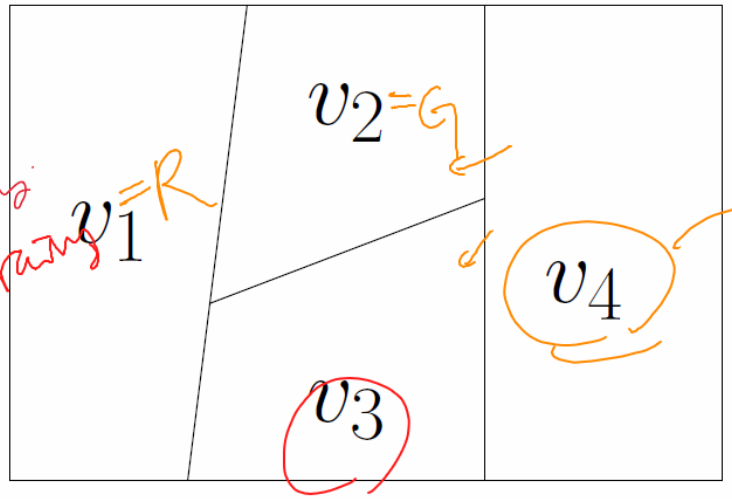
- Least-constraining-value heuristic (value that rules out the smallest number of values for other variables)

Reducing branching factor vs. leaving freedom for future choices.

# CSP Exercise: Map coloring




- Variables: $v_1$, $v_2$, $v_3$, and $v_4$.
- Values: Red, Blue, Green
- Constraints: Color map so that no border shares the same color
- Exercises: Variable selection, Value assignment under different conditions.


# Heuristics for Constraint Satisfaction Problems

General strategies for **variable selection**:

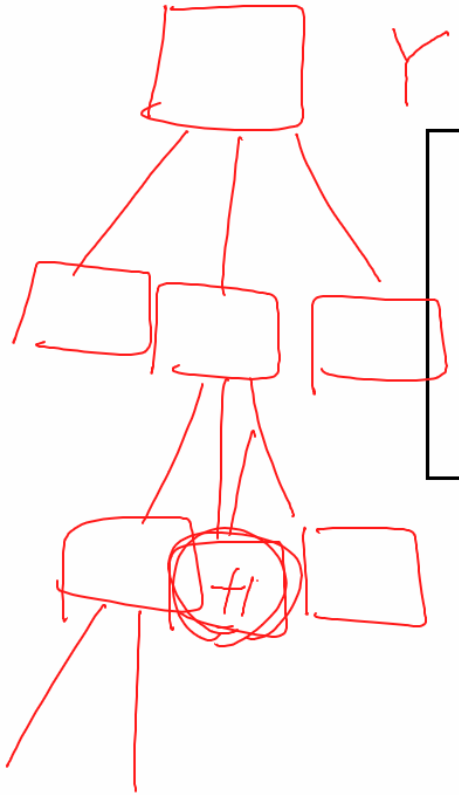
- Most-constrained-variable heuristic (variable with fewest possible values)
- Most-constraining-variable heuristic (variable involved in the largest number of constraints)

and for **value assignment**:

- Least-constraining-value heuristic (value that rules out the smallest number of values for other variables)

Reducing branching factor vs. leaving freedom for future choices.

You stick, and the least constrained value heuristic, and these are for again, variable selection and value. assignment

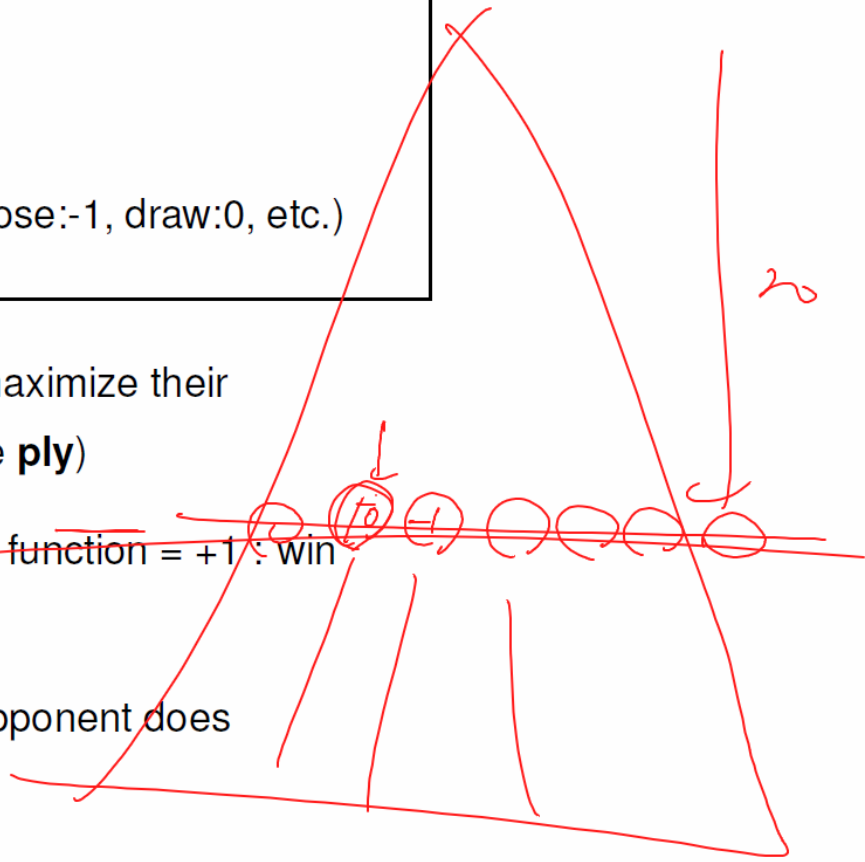
# Two-Person Perfect Information Game

**initial state**: initial position and who goes first

**operators**: legal moves

**terminal test**: game over?

**utility function**: outcome (first player win:+1, lose:-1, draw:0, etc.)



- two players (**MAX** and **MIN**) taking turns to maximize their chances of winning (each turn generates one **ply**)
- one player's victory is another's defeat (utility function = +1 : win for MAX, loss for MIN)
- need a **strategy** to win no matter what the opponent does


# Two-Person Perfect Information Game

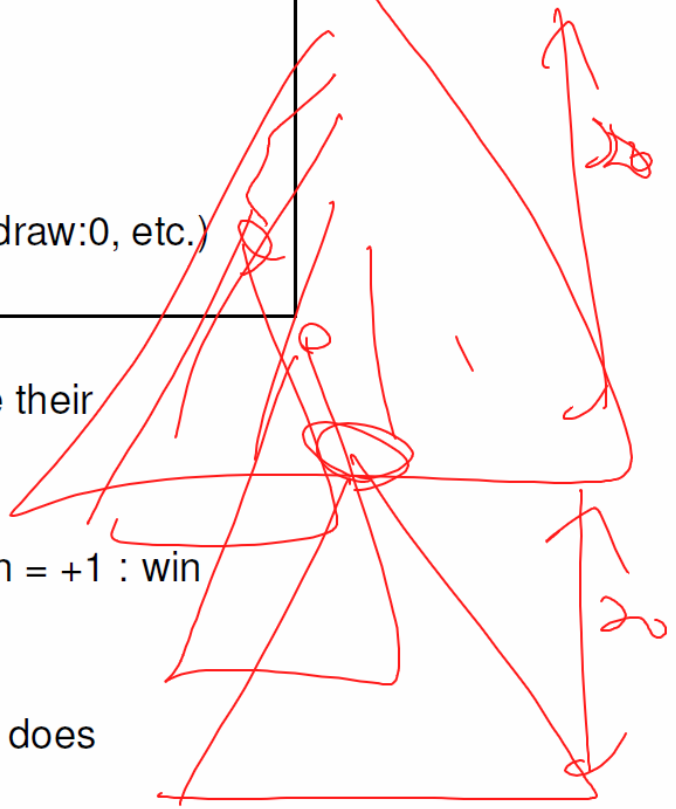
**initial state**: initial position and who goes first

**operators**: legal moves

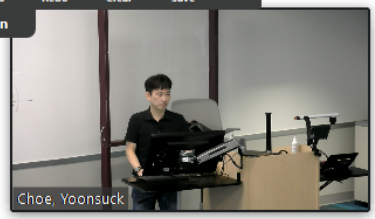
**terminal test**: game over?

**utility function**: outcome (first player win:+1, lose:-1, draw:0, etc.)

- two players (**MAX** and **MIN**) taking turns to maximize their chances of winning (each turn generates one **ply**)
- one player's victory is another's defeat (utility function = +1 : win for MAX, loss for MIN)
- need a **strategy** to win no matter what the opponent does



So, By the way, I mean the on the on the past to this you've been you'll be exploring these different alternatives right




# Two-Person Perfect Information Game

**initial state**: initial position and who goes first

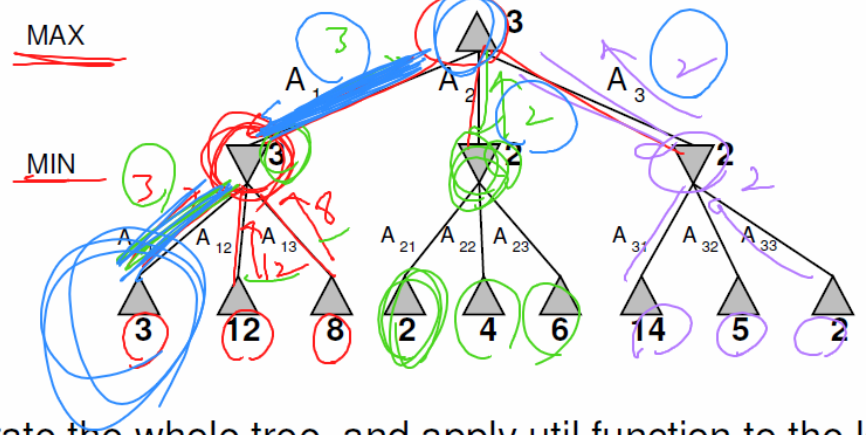
**operators**: legal moves

**terminal test**: game over?

**utility function**: outcome (first player win:+1, lose:-1, draw:0, etc.)

- two players (**MAX** and **MIN**) taking turns to maximize their chances of winning (each turn generates one **ply**)
- one player's victory is another's defeat (utility function = +1 : win for MAX, loss for MIN)
- need a **strategy** to win no matter what the opponent does

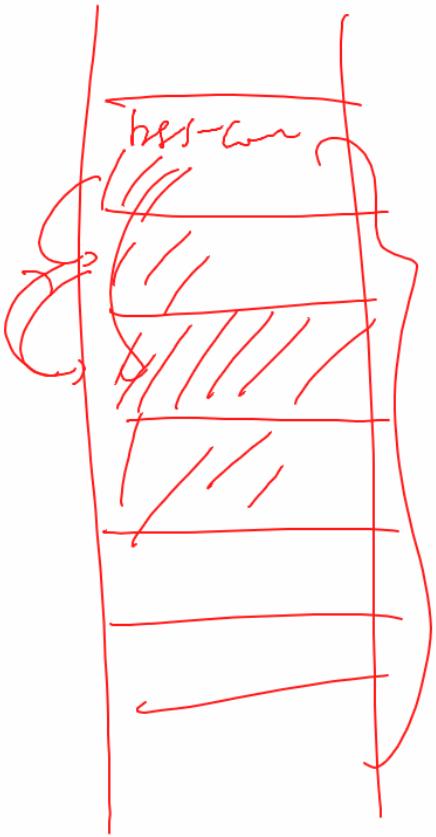

# Minimax: Strategy for Two-Person Perfect Info Games



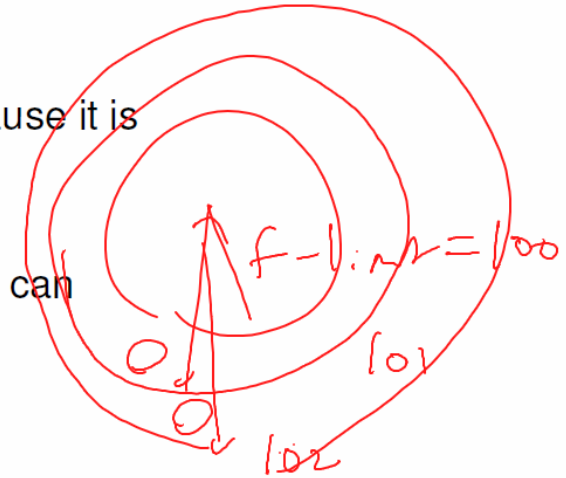

- generate the whole tree, and apply util function to the leaves
- go back upward assigning utility value to each node
- at MIN node, assign **min(successors' utility)**
- at MAX node, assign **max(successors' utility)**
- **assumption**: the opponent acts optimally

Okay, So here, there, the assumption is that you're pulling X. optimally

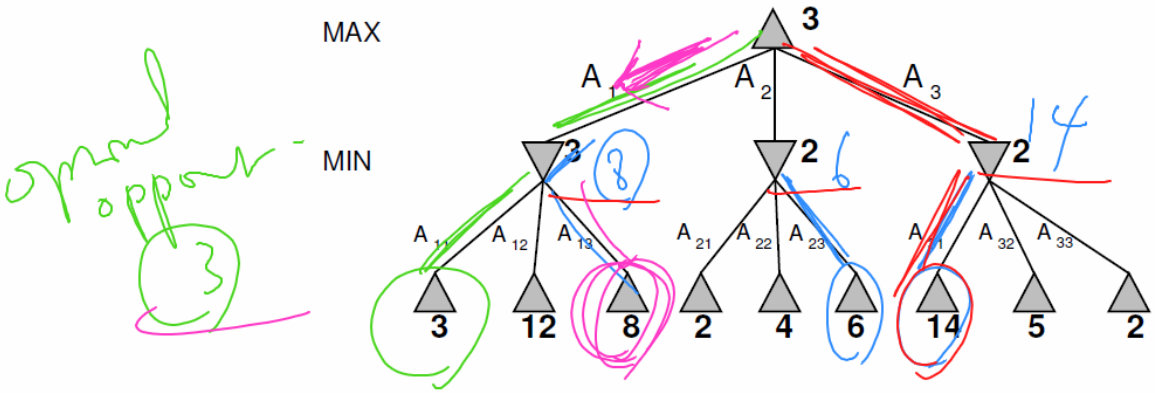
# $IDA^*$: Evaluation



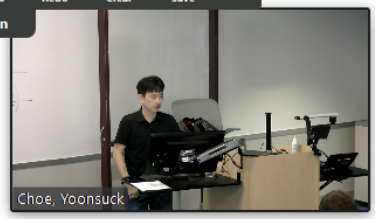
- complete and optimal (with same restrictions as in $A^*$)
- space: proportional to longest path that it explores (because it is depth first!)
- time: dependent on the number of different values $h(n)$ can assume.



# Minimax: Strategy for Two-Person Perfect Info Games




- generate the whole tree, and apply util function to the leaves
- go back upward assigning utility value to each node
- at MIN node, assign **min(successors' utility)**
- at MAX node, assign **max(successors' utility)**
- **assumption**: the opponent acts optimally



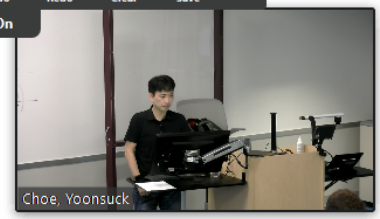

# $IDA^*$: Time Complexity

Depends on the heuristics:

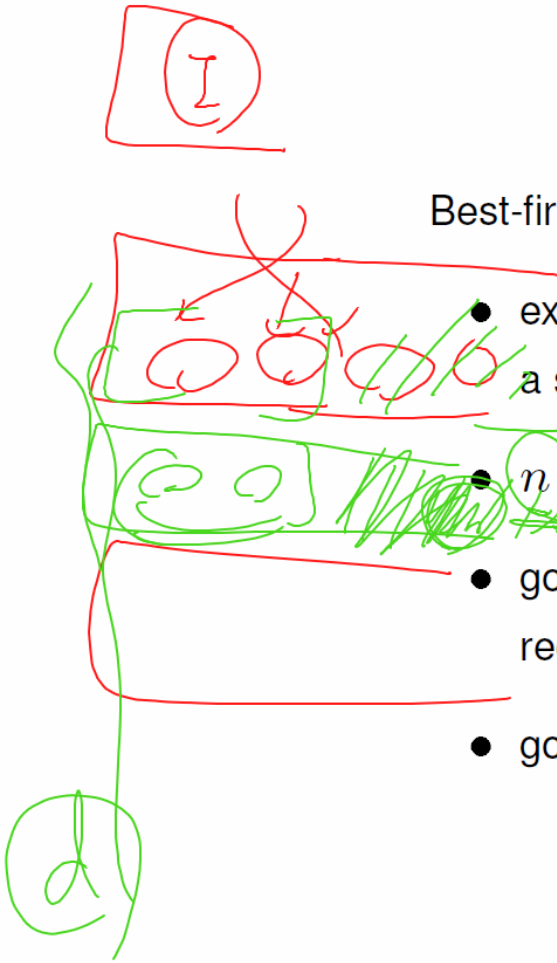
- small number of possible heuristic function values $\rightarrow$ small number of $f$-contours to explore $\rightarrow$ becomes similar to $\mathrm{A}^*$
- complex problems: each $f$-contour only contain one new node

  if $\mathrm{A}^*$ expands $N$ nodes,
  $IDA^*$ expands
  $1 + 2 + .. + N = \frac{N(N+1)}{2} = O(N^2)$
- a possible solution is to have a **fixed** increment $\epsilon$ for the $f$-limit $\rightarrow$ solution will be suboptimal for at most $\epsilon$ ($\epsilon$-admissible)

But the optimality will only suffer by that




# Other Methods: Beam Search



Best-first search with a fixed limited branching factor

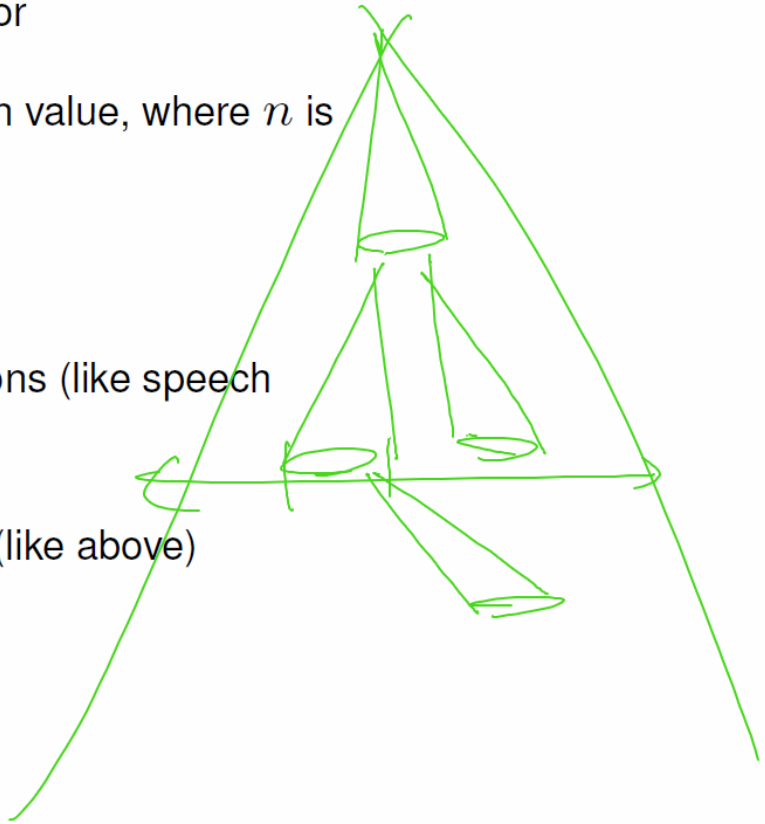
- expand the first $n$ nodes with the best Eval-Fn value, where $n$ is a small number.
- $n$ is called the **width of the beam**
- good for domains with continuous time functions (like speech recognition)
- good for domains with huge branching factor (like above)

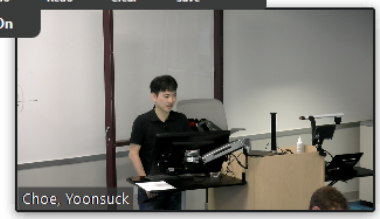
And the rest you want to discard. The choice of this can be critical


# Other Methods: Beam Search

Best-first search with a fixed limited branching factor

- expand the first $n$ nodes with the best Eval-Fn value, where $n$ is a small number.
- $n$ is called the **width of the beam**
- good for domains with continuous time functions (like speech recognition)
- good for domains with huge branching factor (like above)

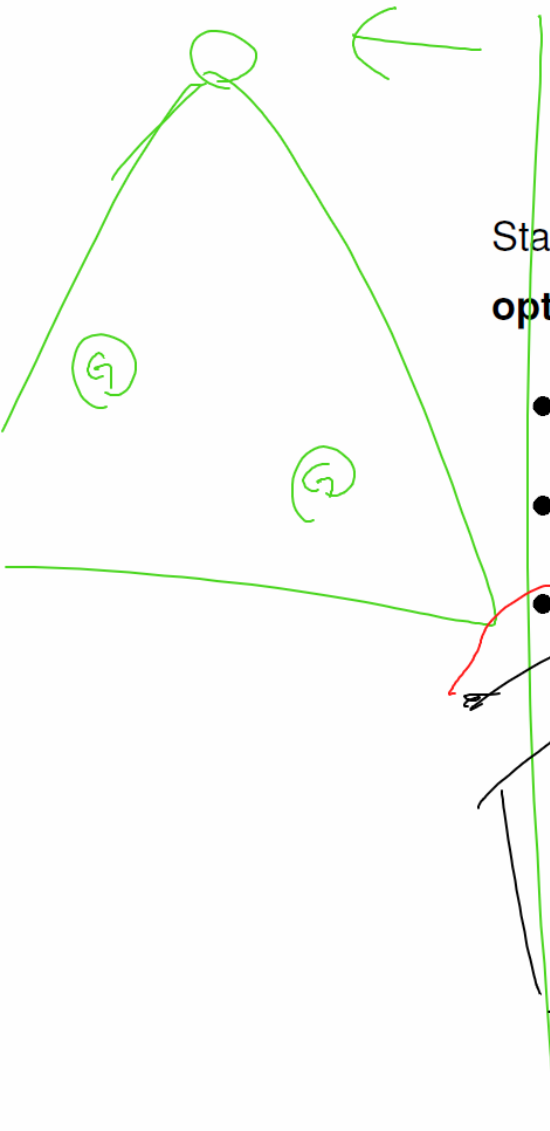


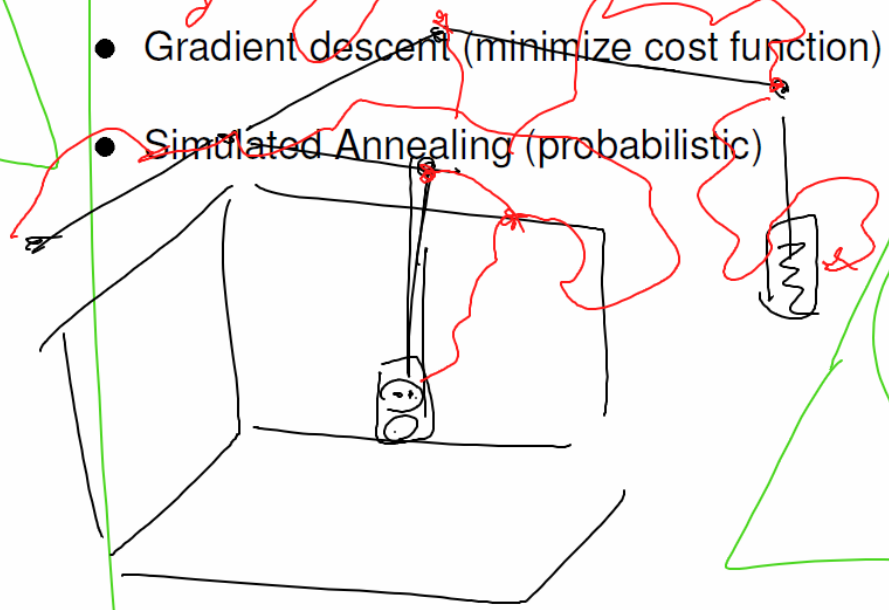


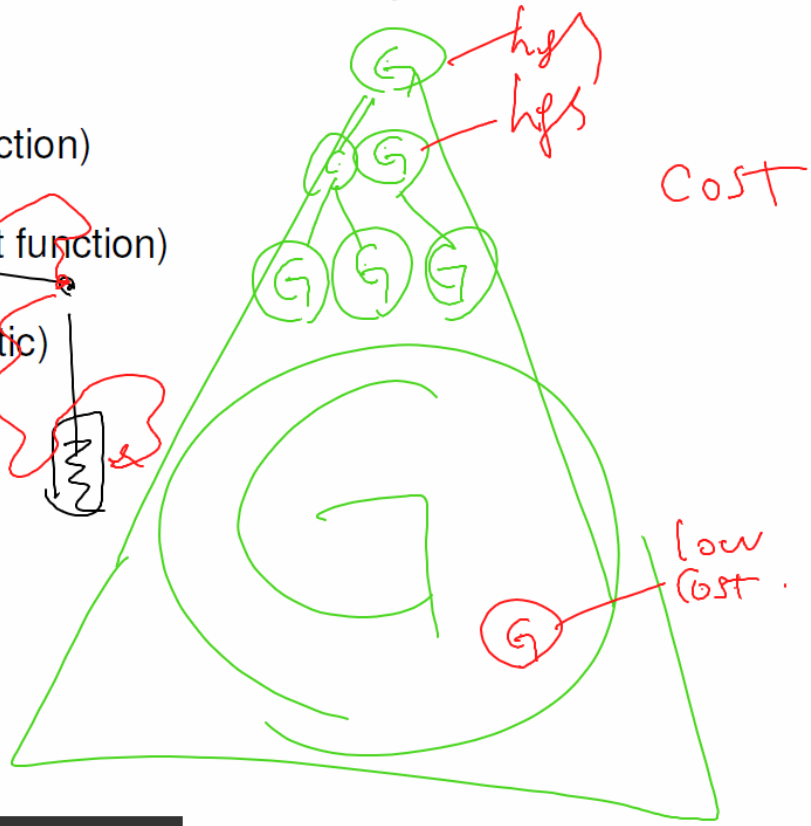

# Iterative Improvement Algorithms

Start with a complete configuration (all variable values assigned, and **optimal**), and **gradually improve** it.

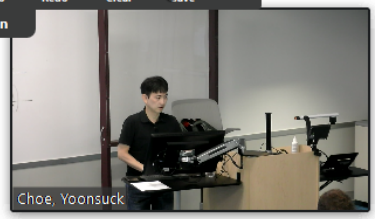
- Hill-climbing (maximize cost function)
- Gradient descent (minimize cost function)
- Simulated Annealing (probabilistic)








Come on right, that's another solution what is it not optimal and probably



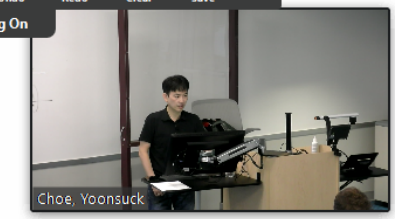

# Iterative Improvement Algorithms

Start with a complete configuration (all variable values assigned, and **optimal**), and **gradually improve** it.

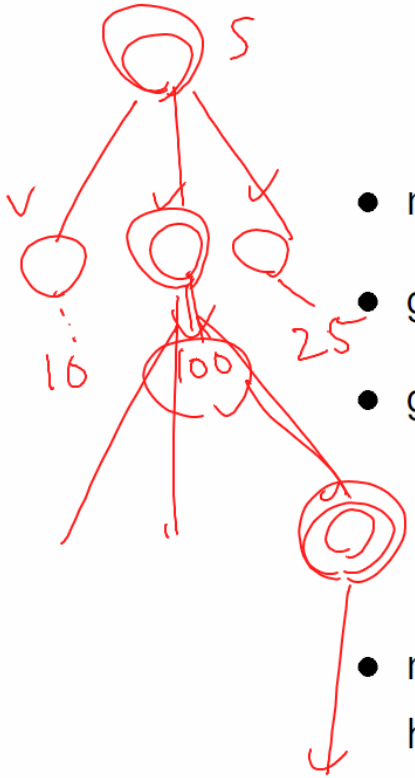
- Hill-climbing (maximize cost function)
- Gradient descent (minimize cost function)
- Simulated Annealing (probabilistic)

$$ income $$ energy

So maximize and minimize that just the the flip side of the coin




# Hill-Climbing




- no queue, keep only the best node
- greedy, no back-tracking
- good for domains where **all nodes are solutions**:
  - goal is to **improve** quality of the solution
  - optimization problems
- note that it is different from greedy best-first search that uses the heuristic function, which keeps a node list