1 Discussion Questions

(a) What is the difference between Forward Checking and AC-3?

Forward checking and AC-3 both enforce arc consistency, but forward checking is more limited. Whenever a variable X is assigned, forward checking enforces arc consistency only for arcs that are pointing to X, which will reduce the domains of the neighboring variables in the constraint graph. Forward checking stops at this point, but AC-3 will continue to enforce arc consistency on neighboring arcs until there are no more variables whose domain can be reduced. As a result, FC ensures arc consistency of the assigned variable and its neighbors only, while AC-3 ensures arc consistency for the whole graph.

(b) Why would one use the following heuristics for CSP?

(i) Minimum Remaining Values (MRV)

MRV: "Which variable should we assign next?"

- Fail fast
- We have to assign all variables at some point, so we might as well do hard stuff first (allowing us to prune the search tree faster/realize we need to backtrack)

(ii) Least Constraining Value (LCV)

LCV: "Which value should we try next?"

- We just want one solution.
- We don't try all combinations of value, so we should try ones that are likely to lead to a solution.

2 CSP: Air Traffic Control

We have five planes: A, B, C, D, and E and two runways: international and domestic. We would like to schedule a time slot and runway for each aircraft to either land or take off. We have four time slots: 1, 2, 3, 4 for each runway, during which we can schedule a landing or take off of a plane. We must find an assignment that meets the following constraints:

- Plane B has lost an engine and must land in time slot 1.
- Plane D can only arrive at the airport to land during or after time slot 3.
- Plane A is running low on fuel but can last until at most time slot 2.
- Plane D must land before plane C takes off, because some passengers must transfer from D to C.
- No two aircrafts can reserve the same time slot for the same runway.

(a) Complete the formulation of this problem as a CSP in terms of variables, domains, and constraints (both unary and binary). Constraints should be expressed implicitly using mathematical or logical notation rather than with words. Make sure to specify variables, domains, and constraints.

Variables: A, B, C, D, E for each plane.
Domains: a tuple (*runway type, time slot*) for runway type $\in$ {international, domestic} and time slot $\in \{1, 2, 3, 4\}$.
Constraints:

$$B[1] = 1$$

$$D[1] \geq 3$$

$$A[1] \leq 2$$

$$D[1] < C[1]$$

$$A \neq B \ \neq C \neq D \neq E$$

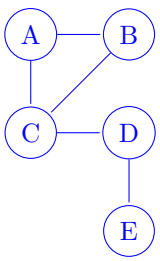
Note here we use B[1] to denote the second value of the tuple assigned to variable B, the time slot value, which is a number in $\{1, 2, 3, 4\}$.

For the following parts, we add the following two constraints:

- Planes A, B, and C cater to international flights and can only use the international runway.
- Planes D and E cater to domestic flights and can only use the domestic runway.

(b) The addition of the two constraints above alters the CSP. Specifically, the domain does not need to include the runway type since this information is carried by the variable, and the binary constraints have changed. Determine the new domain and complete the constraint graph for this problem given the original constraints and the two added ones.

Variables: A, B, C, D, E for each plane.
Domain: $\{1, 2, 3, 4\}$
Constraint Graph:



Explanation of Constraints Graph: We can now encode the runway information into the identity of the variable, since each runway has more than enough time slots for the planes it serves. We represent the non-colliding time slot constraint as a binary constraint between the planes that use the same runways.

(c) What are the domains of the variables after enforcing arc consistency? Begin by enforcing unary constraints. (Cross out values that are no longer in the domain.)

Enforcing arc consistency with AC-3, we have the following domain as a result:

A	~~1~~	2	~~3~~	~~4~~
B	1	~~2~~	~~3~~	~~4~~
C	~~1~~	~~2~~	~~3~~	4
D	~~1~~	~~2~~	3	~~4~~
E	1	2	~~3~~	4

(explanation of process below)
Enforcing unary constraints (in an arbitrary order) first,

1. We cross out 2, 3, 4 from B's domain, adding arcs A → B and C → B to the queue.
2. We cross out 3, 4 from A's domain, adding arcs B → A and C → A to the queue.
3. We cross out 1, 2 from D's domain, adding arcs C → D and E → D to the queue.

Enforcing A → B, we cross out 1 from A's domain; Arcs B → A and C → A are already on the queue.
Enforcing C → B, we cross out 1 from C's domain; add arcs A → C, B → C, and D → C to the queue.
Enforcing B → A, no domain changes are necessary (all values remaining in B's domain have a consistent corresponding value in A's domain); no arcs are added.
Enforcing C → A, we cross out 2 from C's domain; Arcs A → C, B → C, and D → C are already on the queue.
Enforcing C → D, we cross out 3 from C's domain; Arcs A → C, B → C, and D → C are already on the queue.
Enforcing E → D, no domain changes are necessary.
Enforcing A → C, no domain changes are necessary.
Enforcing B → C, no domain changes are necessary.
Enforcing D → C, we cross out 4 from D's domain (there is no c in C's domain such that $c > 4$); add arcs C → D and E → D to the queue.
Enforcing C → D, no domain changes are necessary.
Enforcing E → D, we cross out 3 from E's domain; add arc D → E to the queue.
Enforcing D → E, no domain changes are necessary.
(phew!)

Note: For a general binary CSP, to enforce arc consistency before assigning any variables, you should add all arcs to the initial queue. For this problem, it can be easily seen that if there are no unary constraints, all the arcs will be consistent before any variable is assigned a value. As a result, we can start with the unary constraints and add arcs only for the related variables after enforcing the unary constraints.

(d) Arc-consistency can be rather expensive to enforce, and we believe that we can obtain faster solutions using only forward-checking on our variable assignments. Using the Minimum Remaining Values heuristic, perform backtracking search on the graph, breaking ties by picking lower values and characters first. List the (variable, assignment) pairs in the order they occur (including the assignments that are reverted upon reaching a dead end). Enforce unary constraints before starting the search.

List of (variable, assignment) pairs:

(You don't have to use this table)

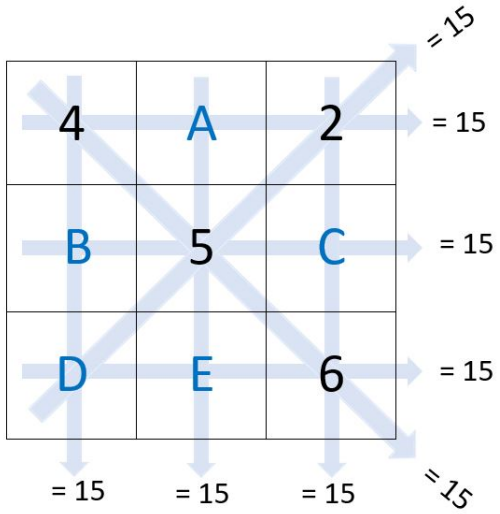
A	1	2	3	4
B	1	2	3	4
C	1	2	3	4
D	1	2	3	4
E	1	2	3	4

Answer: (B, 1), (A, 2), (C, 3), (C, 4), (D, 3), (E, 1)

3 CSP: Magic Square

A magic square is an $n \times n$ grid where each entry is unique and contains one of $\{1, ..., n^2\}$. It has that every row, column, and diagonal sum to the same number.

In this problem, we'll solve a 3×3 magic square by formulating it as a CSP. Each row, column, and diagonal in the 3×3 magic square must sum to 15. We have already filled out some of the numbers for you, but the letters in blue still need to be filled in.



(a) Complete the formulation of this problem as a CSP in terms of variables, domains, and constraints (both unary and binary). Constraints should be expressed implicitly using mathematical or logical notation rather than with words. Make sure to specify variables, domains, and constraints.

Hint: You do not need to create variables for the squares already provided.

Variables: A, B, C, D, E for each empty square

Domain: $\{1, 3, 7, 8, 9\}$

Constraints:

Horizontal:

$$4 + A + 2 = 15 \implies A = 9$$
$$B + 5 + C = 15 \implies B = 10 - C$$
$$D + E + 6 = 15 \implies D = 9 - E$$

Vertical:

$$4 + B + D = 15 \implies B = 11 - D$$
$$A + 5 + E = 15 \implies A = 10 - E$$
$$2 + C + 6 = 15 \implies C = 7$$

Diagonal:

$$D + 5 + 2 = 15 \implies D = 8$$

Other:

$$A \neq B \neq C \neq D \neq E$$

(b) Draw the binary constraint graph for this problem. For simplicity, you may ignore the "alldiff" constraint that all variables are unique when creating this graph.

Explanation of Constraint Graph: We can encode the binary constraints as edges between pairs of nodes. Note that in this problem, all rows, columns and diagonals have at most 2 variables, since at least one number is already given. Therefore, every pair of variables in the same row, column, or diagonal has the binary constraint that the sum of those two variables along with the third given number must be 15.

Here are the four binary constraints (represented by edges in the graph):

- A is in the same column as E, so A and E have a binary constraint.
- E is also in the same row as D, so E and D have a binary constraint.
- D is also in the same column as B, so D and B have a binary constraint.
- B is also in the same row as C, so B and C have a binary constraint.

(c) Use the binary constraint graph to run the AC-3 algorithm and find a solution to the magic square.

Enforcing unary constraints (in an arbitrary order) first,

(a) We cross out 1, 3, 7, 8 from A's domain as it is already fully constrained.

(b) We cross out 1, 3, 8, 9 from C's domain as it is already fully constrained.

(c) We cross out 1, 3, 7, 9 from D's domain as it is already fully constrained.

AC3 Solution:

Enforcing arc consistency with AC-3, we have the following domain as a result:

A	~~1~~	~~3~~	~~7~~	~~8~~	9
B	~~1~~	3	~~7~~	~~8~~	~~9~~
C	~~1~~	~~3~~	7	~~8~~	~~9~~
D	~~1~~	~~3~~	~~7~~	8	~~9~~
E	1	~~3~~	~~7~~	~~8~~	~~9~~

Our starting queue will contain, in an arbitrary but convenient order, arcs E → A, B → C, B → D, and E → D.

Enforcing E → A, we cross 3, 7, 8, 9 from E's domain. Arcs A → E and D → E will be added to the queue.
Enforcing B → C, we cross 1, 7, 8, 9 from B's domain, Arcs D → B and C → B will be added to the queue.
Enforcing B → D, no domain changes are necessary.
Enforcing E → D, no domain changes are necessary.
Enforcing A → E, no domain changes are necessary.
Enforcing D → E, no domain changes are necessary.
Enforcing D → B, no domain changes are necessary.
Enforcing C → B, no domain changes are necessary.

(d) How would the representation of this problem change if we had three variables in a singular row?

We would have a ternary constraint.

4 MRV and LCV in Action

Raashi, Simrit, and Ihita want to paint "15-281" on the fence tomorrow in honor of their favorite class. They have the following paint collections at home:

- Raashi = Red, Yellow, Green
- Simrit = Red, Yellow, Pink
- Ihita = Red, Pink

Each of them will contribute exactly one bucket of paint from their collections such that no two TAs bring the same paint color.

Raashi suddenly remembers going over CSPs in lecture, and suggests formulating this problem as a CSP to determine an assignment of each TA to a paint color so that all three chosen paint colors are different. Simrit isn't fully convinced yet that LCV and MRV will help speed up a constraint satisfaction problem, so Raashi asks for your help to convince Simrit.

(a) Let's use the minimum remaining values (MRV) and least constraining value (LCV) heuristics to assign TAs to paint colors. Recall that the MRV heuristic determines which *variable* to assign, while the LCV heuristic determines which *value* to assign to that variable to. How many times will we backtrack to a previous assignment? Assume we break ties in rainbow order.

We will backtrack zero times.

We first use the MRV (minimum remaining values) heuristic to choose which TA gets assigned next. Ihita has 2 possible paint colors, while Raashi and Simrit have 3. Since Ihita has the fewest possible paint colors left, we will assign her next.

We will assign Ihita a paint color using the LCV heuristic.

If we assign Ihita to "Pink," the sum of the number of paint colors remaining over the other TAs is 5.

- Raashi = Red, Yellow, Green
- Simrit = Red, Yellow, ~~Pink~~
- Ihita = ~~Red~~, **Pink**

If we assign Ihita to "Red," the sum of the number of paint colors remaining over the other TAs is 4.

- Raashi = ~~Red~~, Yellow, Green
- Simrit = ~~Red~~, Yellow, Pink
- Ihita = **Red**, ~~Pink~~

Since 5 ¿ 4, assigning Ihita to "Pink" is the least constraining value (or paint color).

Now, we will use MRV again to choose which TA will get assigned next. Simrit has 2 possible paint colors, while Raashi has 3. Since Simrit has the fewest possible paint colors left, we will assign her next.

If we assign Simrit "Red", there are 2 paint colors left for Raashi.

- Raashi = ~~Red~~, Yellow, Green

- Simrit = **Red**, ~~Yellow~~, ~~Pink~~
- Ihita = ~~Red~~, **Pink**

If we assign Simrit "Yellow", there are 2 paint colors left for Raashi.

- Raashi = Red, ~~Yellow~~, Green
- Simrit = ~~Red~~, **Yellow**, ~~Pink~~
- Ihita = ~~Red~~, **Pink**

Since we have a tie (and we break ties in rainbow order), we will assign Simrit to "Red."

Raashi is the only TA left without a paint color assignment. We can assign her either "Yellow" or "Green," but choose "Yellow" because we break ties in rainbow order.

We have satisfied the constraints (that no two TAs bring the same paint color) without backtracking to previous assignments.

(b) Suppose we use a new set of heuristics to assign TAs to paint colors: maximum remaining values (instead of MRV) and most constraining value (instead of LCV). How many times will we backtrack to a previous assignment?

We will backtrack one time.

We first use the maximum remaining values heuristic to choose which TA gets assigned next. Ihita has 2 possible paint colors, while Raashi and Simrit have 3. Since Raashi and Simrit have the most possible paint colors left, we will assign Raashi next (tiebreaking by order).

We will assign Raashi a paint color using the most constraining value heuristic.

If we assign Raashi to "Red," the sum of the number of paint colors remaining over the other TAs is 3.

- Raashi = **Red**, ~~Yellow~~, ~~Green~~
- Simrit = ~~Red~~, Yellow, Pink
- Ihita = ~~Red~~, Pink

If we assign Raashi to "Yellow," the sum of the number of paint colors remaining over the other TAs is 4.

- Raashi = ~~Red~~, **Yellow**, ~~Green~~
- Simrit = Red, ~~Yellow~~, Pink
- Ihita = Red, Pink

If we assign Raashi to "Green," the sum of the number of paint colors remaining over the other TAs is 5.

- Raashi = ~~Red~~, ~~Yellow~~, **Green**
- Simrit = Red, Yellow, Pink
- Ihita = Red, Pink

Since $3 < 4$ and $3 < 5$, we will assign Raashi to "Red" using the most constraining value heuristic.

Now, we will use the *maximum* remaining values heuristic again to choose which TA gets assigned next. Simrit has 2 possible paint colors, while Ihita has 1. Since Simrit has more possible paint colors left, we will assign her next.

If we assign Simrit to "Yellow," the sum of the number of paint colors remaining over the other TAs is 1.

- Raashi = **Red**, ~~Yellow~~, ~~Green~~
- Simrit = ~~Red~~, **Yellow**, ~~Pink~~
- Ihita = ~~Red~~, Pink

If we assign Simrit to "Pink," the sum of the number of paint colors remaining over the other TAs is 0.

- Raashi = **Red**, ~~Yellow~~, ~~Green~~
- Simrit = ~~Red~~, ~~Yellow~~, **Pink**
- Ihita = ~~Red~~, ~~Pink~~

Since $0 < 1$, we will assign Simrit to "Pink" using the *most* constraining value heuristic.

Now, Ihita has no paint colors to choose from, so we have not found a solution to our CSP.

Thus, we must backtrack and assign Simrit to "Yellow" instead of "Pink."

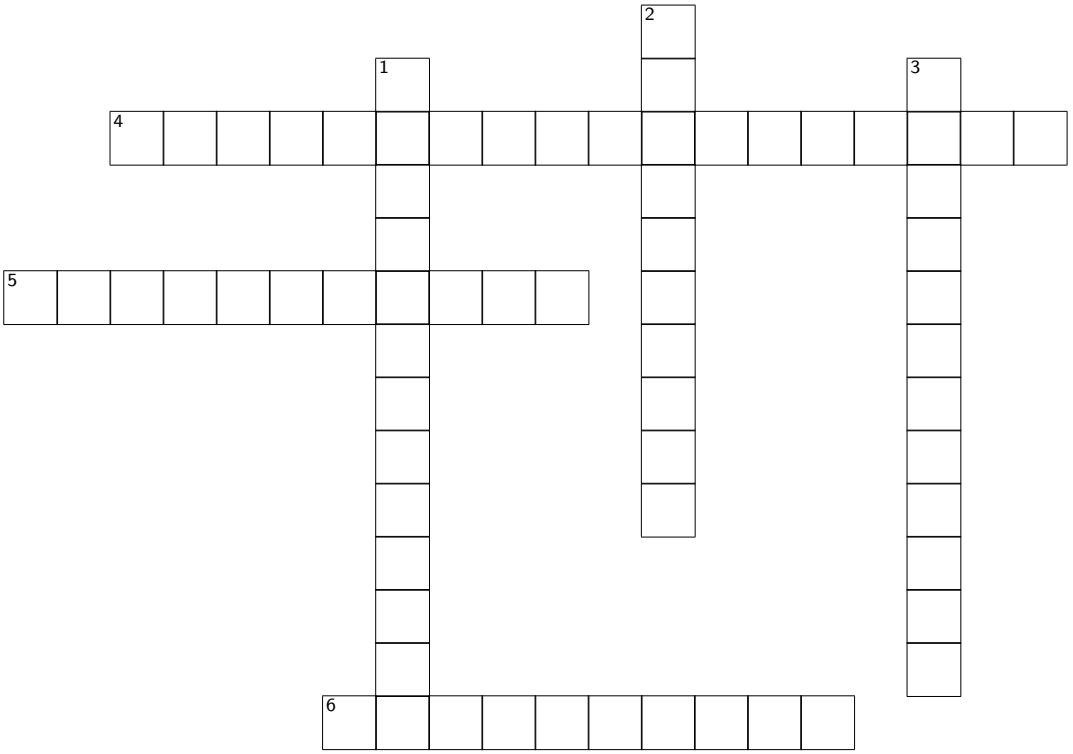
- Raashi = **Red**, ~~Yellow~~, ~~Green~~
- Simrit = ~~Red~~, **Yellow**, ~~Pink~~
- Ihita = ~~Red~~, Pink

Ihita is the only TA left without a paint color assignment, and there is only one paint color left in her domain. Thus, we can assign Ihita to "Pink."

We have satisfied the constraints (that no two TAs bring the same paint color), but we had to backtrack to a previous assignment one time.

5 Missing in the Mountains

The 15-281 Course Staff decided to climb the Rocky Mountains together over Winter Break. On their way back down from the mountains, they realize they left Simrit at the top of the tallest mountain! They have no idea where on the mountain they are or which mountain they are on, and are worried about how they will find Simrit before lecture. Help the 281 Staff remember all of the local search algorithms they have learned so they can save Simrit!

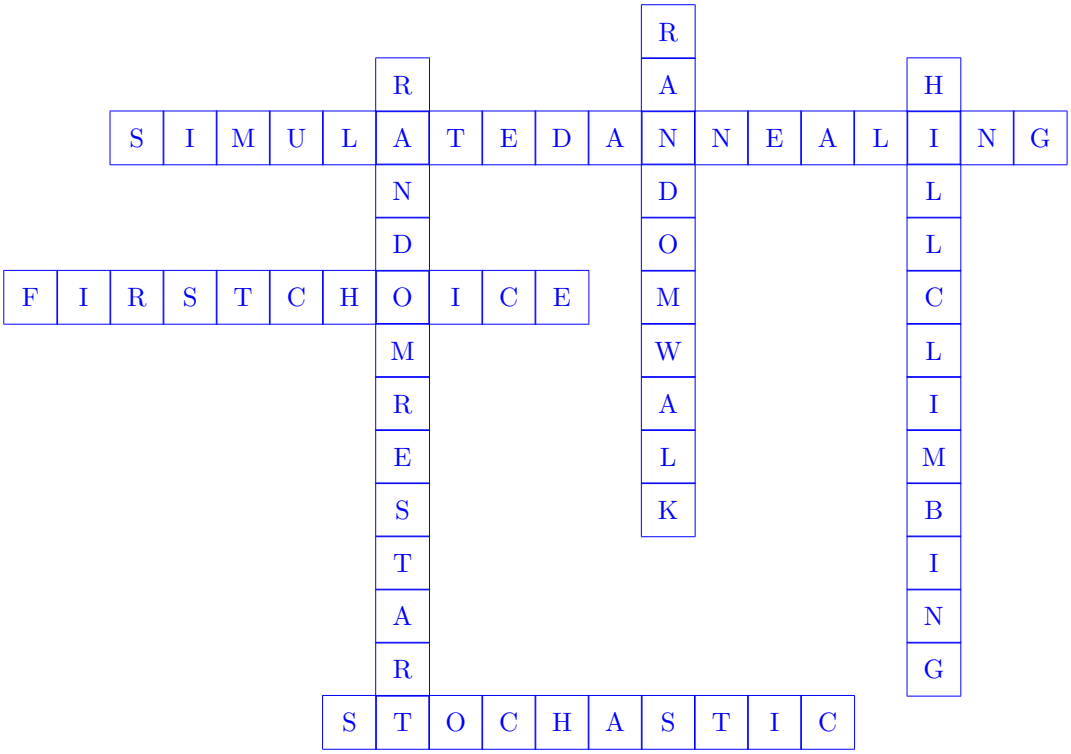


Down

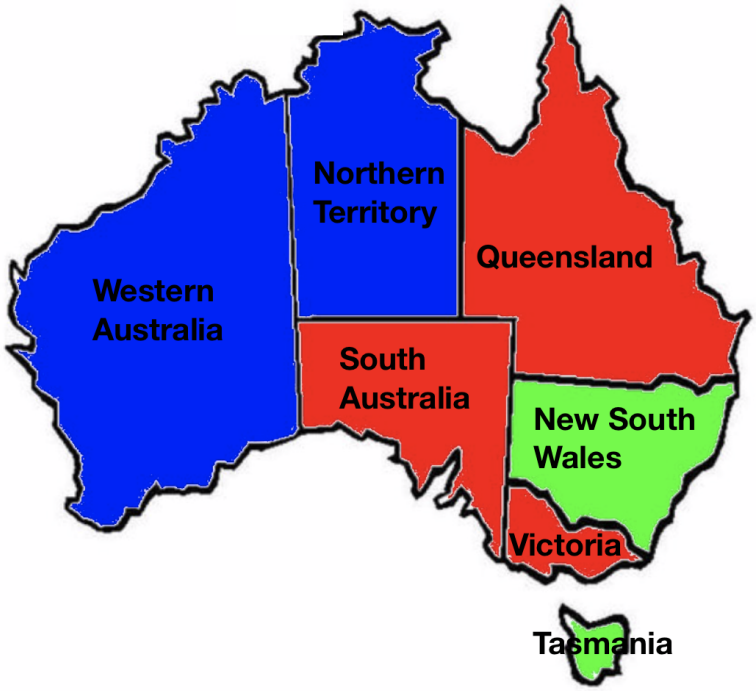
1. A variant of hill-climbing where you conduct a series of searches from randomly generated starting states until the goal is found.
2. A local search technique where you uniformly randomly choose a neighbor to move to.
3. A type of greedy local search where you move uphill to local maxima.

Across

4. A local search technique where you allow for downhill moves but make them rarer as time goes on.
5. A variant of hill-climbing where you generate successors randomly (one by one) until a better one is found.
6. A variant of hill climbing in which you choose a move randomly from the uphill moves, with the probability of a move being chosen dependent on the "steepness" (amount of improvement from making that move).


R
R A H
SIMULATEDANNEALING
N D L
D O L
FIRSTCHOICE M C
M W L
R A I
E L M
S K B
T I
A N
R G
STOCHASTIC

6 Map Coloring with Local Search



Recall the various local search algorithms presented in lecture. Local search differs from previously discussed search methods in that it begins with a complete, potentially conflicting state and iteratively improves it by reassigning values. We will consider a simple map coloring problem, and will attempt to solve it with hill climbing.

(a) How is the map coloring problem defined (In other words, what are variables, domain and constraints of the problem)? How do you define states in this coloring problem?

- Variables: WA, NT, SA, Q, NSW, V, T (States in Australia)
- Domain: Green, Red, Blue
- Constraints: Adjacent countries can't have the same color assignment. e.g: Implicit: $WA \neq NT$
  Explicit: (WA, NT) $\in$ (red, blue), (red, green), (blue, red), (blue, green), (green, red), (green, blue)
- Problem state: a full coloring of the map (i.e., color assignments to all variables).

(b) Given a complete state (coloring), how could we define a neighboring state?

A neighboring state could be a full coloring of the graph with a different color assignment to only one variable.

(c) What could be a good heuristic be in this problem for local search? What is the initial value of this heuristic?

The heuristic could be the number of variable pairs that have conflicting colors. In the initial state, the following 3 pairs (WA-NT, Q-SA, SA-V) are conflicting, so the heuristic h = 3. (Note: there could be other possible heuristics for this problem.)

(d) Use hill climbing to find a solution based on the coloring provided in the graph.

Let h be our heuristic value.
In the original graph, we have 3 coloring conflicts as stated in (c). Depending on the search order, the assignment order might be different and the searched path lengths as well as coloring can also vary. We represent the coloring of states in a list with the following order: [WA, NT, Q, SA, NW, V, T]. Below are two examples of potential search paths.

- Step 1: h = 3. Conflicts are WA-NT, Q-SA, SA-V. We start with WA-NT. Coloring NT with Green would resolve the WA-NT conflict. Coloring: [B, G, R, R, G, R, G]
- Step 2: h = 2. Conflicts are Q-SA, SA-V. We can pick SA-Q pair and assign Blue to SA, which would resolve SA-Q and SA-V conflicts but will add coloring conflict for WA-SA pair. Still, it decreases the number of conflicts and is a better neighboring state. Coloring: [B, G, R, B, G, R, G]
- Step 3: h = 1. Conflict is just WA-SA pair. We can simply assign WA with Red to resolve this conflict, where we completed the search and found a solution to the problem. Coloring: [R, G, R, B, G, R, G]

We got pretty lucky in the search above and found a solution in 3 steps. However, local search may not always resolve conflicts optimally. Below is an example where it has to resolve the conflicts with more steps.

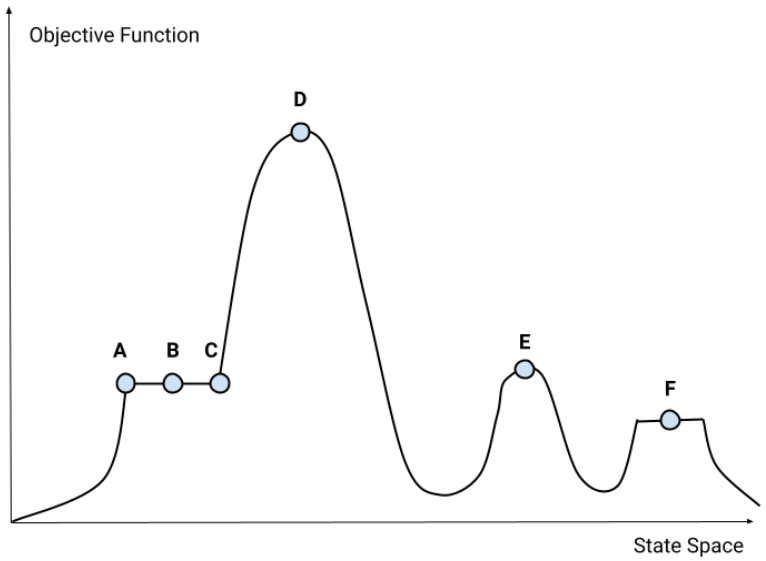
- Step 1: h = 3. Conflicts are WA-NT, Q-SA, SA-V. We start with WA-NT. Coloring WA with Green would resolve the WA-NT conflict. Coloring: [G, B, R, R, G, R, G]
- Step 2: h = 2. Conflicts are Q-SA, SA-V. We can resolve both of these conflicts by assigning Blue to SA, which will lead us to a better neighboring state with only one conflict: NT-SA. Coloring: [G, B, R, B, G, R, G]
- Step 3: h = 1. Conflicts are NT-SA. However, looking at all possible assignments to both of these two states, we see that no matter what color we assign to either one of them, we can't find a better neighboring state. Therefore the current iteration of hill climbing would end and we would restart from the initial state if we are applying random-restart hill climbing. An alternative is to use simulated annealing, which would allow us to sometimes move to states of higher heuristic value in order to escape local minima.

We see that in the second search, we need more steps to complete search. There are other possible search steps sequences depending on the choice on the order of conflicts to resolve, and the color assignment when resolving each conflict.

(e) How is local search different from tree search?

Tree search has a frontier while local search does not. Local search also never backtracks if it gets stuck.

7 Local Search Discussion Questions



Consider the state space above in the context of local search. Recall that our goal is to find the state that maximizes the objective.

(a) Consider the points A, B, C, D, E, and F on the graph.

(i) Which of the points on the graph are on a shoulder? Which of those points are local maximums?

A, B, and C are on a shoulder. A and B are local maximums, but C is not.

A is considered a local maximum since it does not have a neighbor with better objective value. We can use the same reasoning for B.

C is not considered a local maximum because it has a neighbor to the right with a better objective value.

(ii) Which of the points on the graph are a "flat" local maximum?

B and F are on a "flat" local maximum.

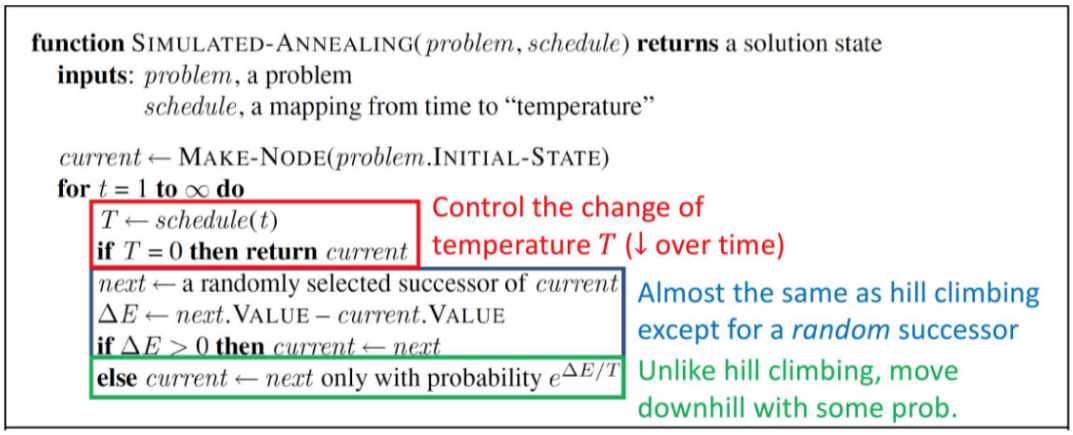
(iii) What is the difference between a shoulder and a "flat" local maximum?

The key difference between a "flat" local maximum and a shoulder is that there is no uphill exit from a flat local maximum, whereas from a shoulder, uphill progress is technically possible from one of the endpoints of the shoulder.

(b) Let's take a look at simulated annealing. Simulated Annealing is quite similar to hill climbing.

- Instead of picking the best move, it picks a random move.
- If the move improves the situation, the move is always accepted.

- Otherwise, it accepts the move with some probability less than 1



(i) How does the sign of ΔE reflect the "badness" of a move?

ΔE is negative for a "bad" move.

(ii) In simulated annealing, we control the temperature T. How does the value of T impact the probability with which we choose a "bad" move?

0.5in Recall that the probability of choosing a "bad move" is $\frac{1}{e^{|\Delta E|/T}}$ since ΔE is negative for a bad move. For smaller values of T, our denominator is larger, so the probability of choosing a bad move is low. For larger values of T, our denominator is smaller, so the probability of choosing a bad move is high.

(c) Mark True or False for each of the following statements.

(i) Regular hill climbing is optimal (i.e., will always find the global maximum)

False. Regular hill climbing, a.k.a. greedy local search, is not optimal since it may choose a local optima as the solution.

(ii) Random restart hill climbing is optimal when given an infinite amount of time.

True. Random restart hill climbing is optimal since it will eventually restart at an initial state that allows it to find the global optimum.

(iii) Simulated annealing allows for downward moves according to some fixed constant temperature T.

False. Simulated annealing allows for downhill moves according to a decreasing temperature T in order to make them rarer as time goes on.

(iv) Simulated annealing is generally less time efficient than random walk.

False. Simulated annealing combines the efficiency of hill climbing with the completeness of random walk, making it generally much faster than random walk on its own.

(v) A random walk algorithm is more likely to choose a better neighbor than a worse one.

False. A random walk search does not consider the optimality of the neighbors at all.