Local Search Guide

What is Local Search?

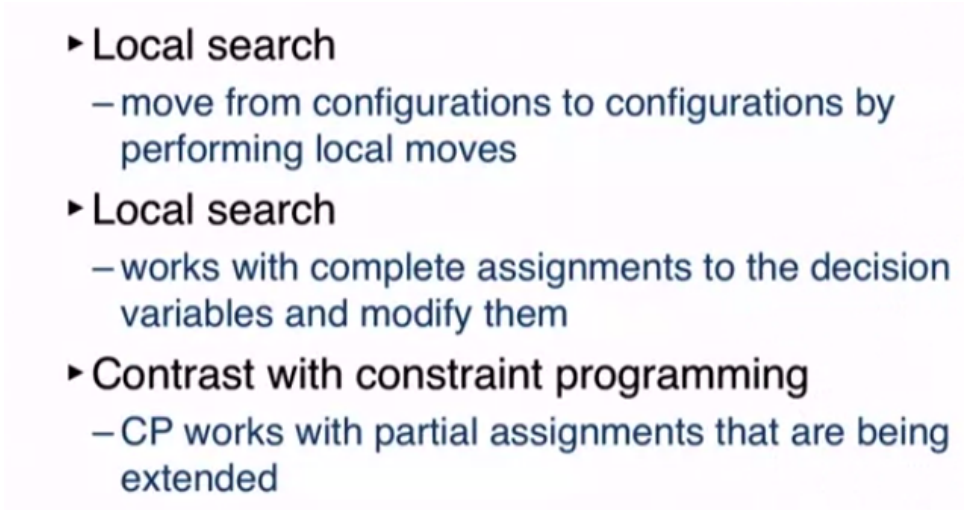
Local search is probably the oldest and most intuitive optimization technique. It consists in starting from a solution and improving it by performing (typically) local perturbations (often called moves). Local search has evolved substantially in the last decades with a lot of attention being devoted on which moves to explore. These notes are the ones we made after extensively studying local search from various sources: the the first exposure to LOCAL SEARCH via Pascal van Hatenryck Coursera Lectures and then exploring local search techniques in several research papers, online sources etc.

Types of problems

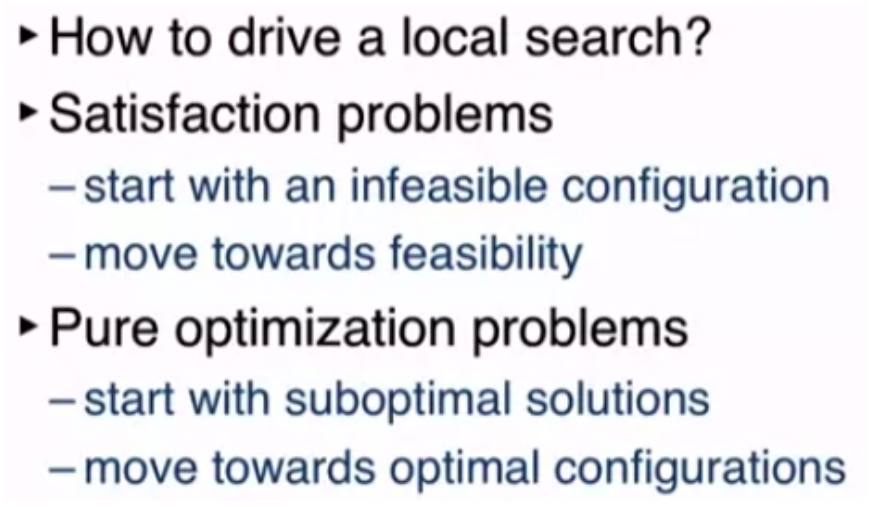
We can deal with 3 main types of problems:

1. **Satisfaction problems:** We have some constraints and we start with a config which doesn't satisfy the constraints and we gradually move towards satisfiability
2. **Pure optimization problems:** We have a function which we want to optimize and we move towards the direction of optimization
3. **Constrained optimization:** This is the tougher class which needs to satisfy constraints as well as optimize a function at the same time

In simple words, local search is a technique that explores the space of possible solutions in sequential fashion, moving from a current solution to a “nearby” one.

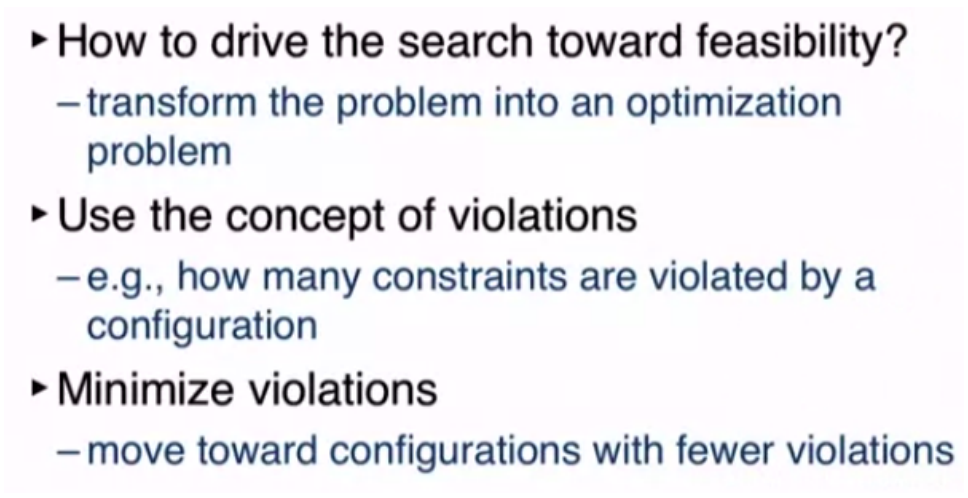


Some local search assignments can thus violate the ‘constraints’ that you’d put in a constraint programming solution, unlike constraint programming where all intermediate states satisfy all constraints.



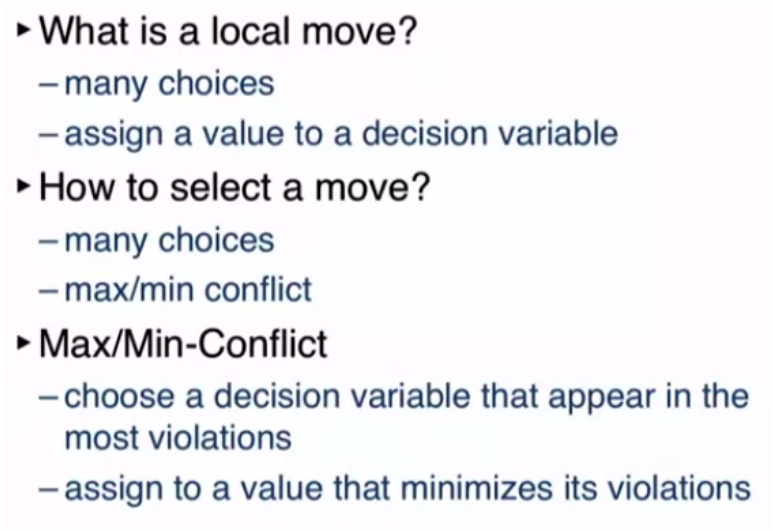
1.1 Satisfiability to Optimization

- Goal is to transform the satisfaction problem to an optimization problem
- Might try to minimise violations, assign a penalty for each violation
- In local search, all states are possible solutions to the problem. The solution may be good or bad. Eg: In an 8-queen problem, all the states which we will traverse through in local search wilutions (here, it means that they have same value for most of the decision variables).I have 8 queens on the board. So, instead of first placing 1st queen on the board, then 2nd and so on, we directly start with a solution irrespective of whether it is a high quality solution or a low quality solution.
- The intuition behind the local move is that two states which are neighbours have relatively similar sol



How constraint violations are counted can differ, one could even build an arbitrary loss function on it with different weights to different violations.

What is meant by a local move



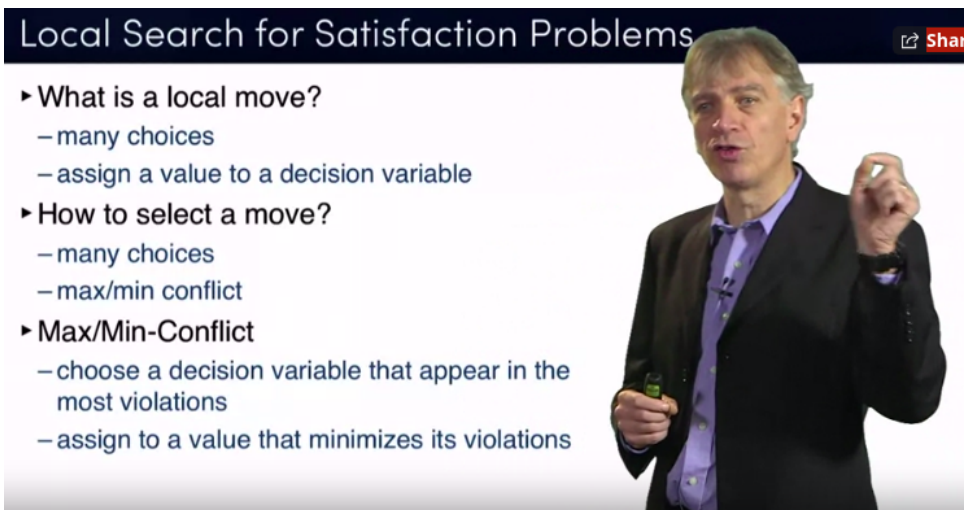
In the N-Queens problem, your local move can be ‘move along the current row’ because we know one queen will be there in every column, and queens are not distinct. This reduces our ‘search space’.

- **There are many choices to move from one config to another.**
- **Simple thing to do would be to select a decision variable and to change its value**

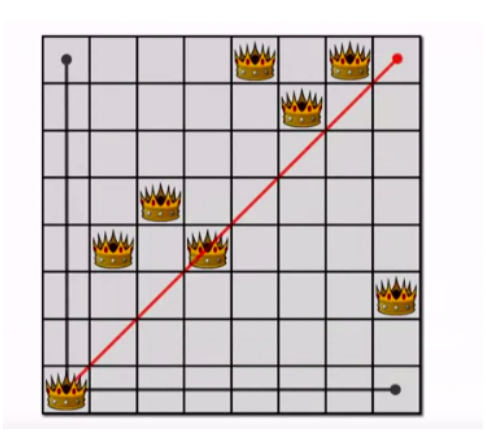
How to select a move to the next neighbour:

Approach 1: Max/min conflict

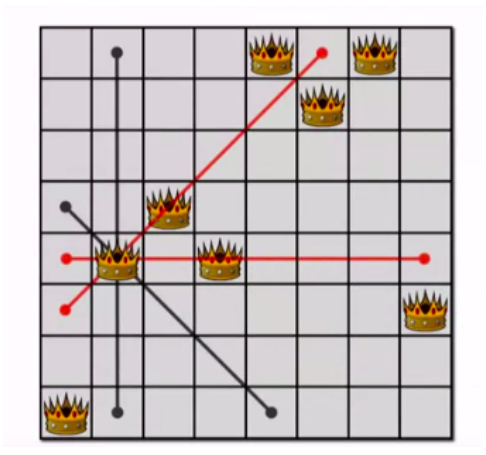
- Choose the variable which is present in most number of violations and assign to it a value that minimizes the new number of violations
- Assign it to a value that minimizes its violations

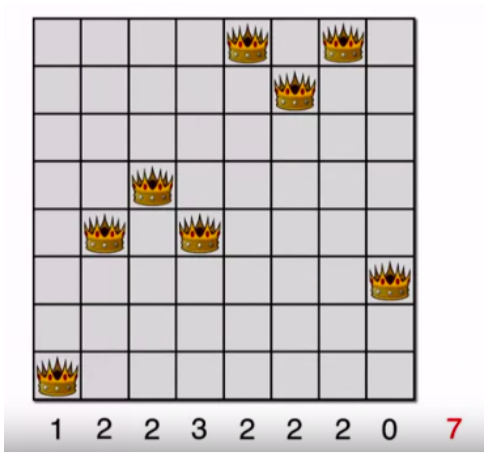


We see that queen at column 1 (the queen at A1) selected is in just a single violation (the queen at D4)



We see that 2nd queen (the one at B4) is in several violations

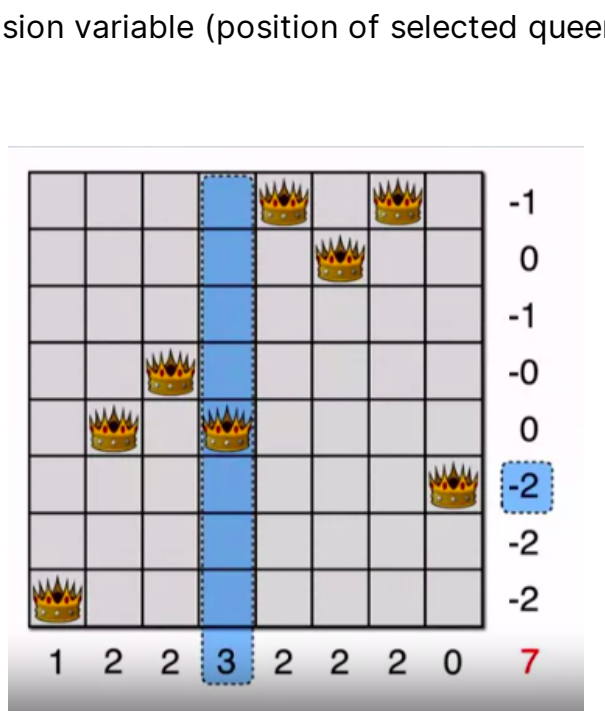




In above picture, the numbers below the respective columns signifies number of violations the queen in that column appears in and the red digit is the total number of violations

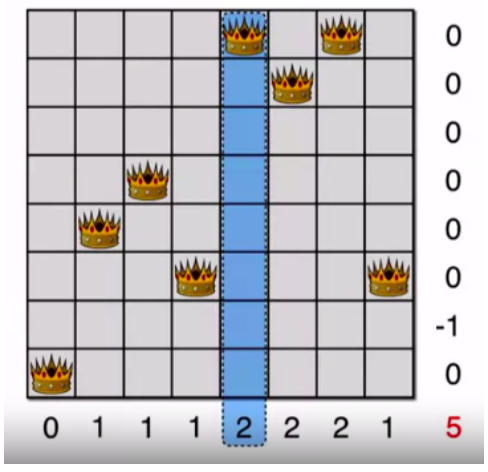
Step 1: Choose queen which appears in max number of violations (Here, queen at D4)

Step 2: Choose the new value of decision variable (position of selected queen) so as to minimize the new number of violations



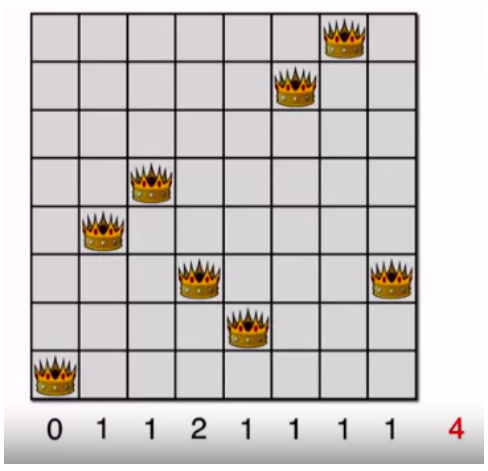
The values at the right in the above picture is the difference **(**$projected\ violations\ if\ shifted\ to\ this\ row - current\ violations$**).** Since -2 appears thrice (ie for a move to 3 rows) , we may choose either of the three options.

- Below is the new config after a local move (the queen at D4 moved to D3):

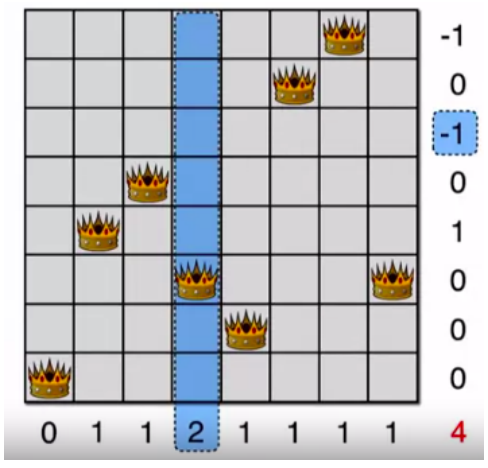


Now we can start repeating the steps again. Here, we see that queens in columns E,F and G are a part of maximum number of violations. Let's say we choose to move the queen at $E8$ to a suitable position. The difference **(**$projected\ violations\ if\ shifted\ to\ this\ row - current\ violations$**)** can be seen to the rightmost end of each row.

Moving the queen to $E2$ is the only move where moving queen at $E8$ can lead to reduction in total number of violations and hence $E2$ is an easy choice.

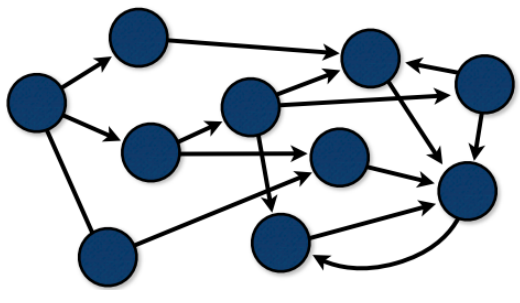


Now, in above configuration, the queen at D3 is the one which is occurring in most number of violations and hence, is chosen as the next queen to be moved. We can repeat the above set of steps continuously until we finally converge to a configuration with no violations



Definition of a local minima and visualizing neighbourhood as a graph

- Optimizing a function *f*
- Local moves define a neighborhood
  - configurations that are close
  - N: C →2^C
- Local search is a graph exploration



In the graph, nodes are ‘states’ or ‘assignments’ and edges are between those assignments which we can transition between in 1 move. Directed graph as it may not make sense to go from a ‘lower loss function’ assignment to a higher one in some cases. Can always add backward edge if needed anyway. The algorithm designer decides which edges should be allowed for each node. Lower the edges, there might be less chance of reaching optimal solution (not until a certain point though) but also algorithm will run faster.

Local Minima: Every possible move leads to an equal or worse configuration.

C =current configuration

$N(C)$ =set of configurations I can move to from the current configuration (neighbourhood of the current configuration)

$f(C)$ =value of objective function for configuration C

Then a configuration C is called a point of local minima if $\forall$ **n** $\in N(C) : f(n) \geq f(C)$

- Local minima
  - a configuration c is a local minima with respect to neighborhood N if

$$\forall\, n \in N(c) : f(n) \geq f(c)$$

- Local search

```
select a configuration c;
I = { n in N(c) | f(n) < f(c) };
while (|I| > 0) {
   c = select a configuration from I;
   I = { n in N(c) | f(n) < f(c) };
}
```

No guarantees for achieving global optima. The local minima may also be high quality (close to global minima) or bad quality(far from global minima). Escaping from a bad local optima is a critical issue.

> If you want guarantees, buy a toaster (C. Eastwood)

Local Search for Satisfaction

- The function *f* minimizes the constraint violations
- When *f(c) = 0*, the configuration *c* is feasible
- Constraint violation
  - 0/1: whether the constraint is violated
  - degree of violation
- Variable violations
  - e.g., the number of violated constraints the variables appears in

Car sequencing problem statement

Formal description

A number of cars are to be produced; they are not identical, because different options are available as variants on the basic model. The assembly line has different stations which install the various options (air- conditioning, sun-roof, etc.). These stations have been designed to handle at most a certain percentage of the cars passing along the assembly line. Furthermore, the cars requiring a certain option must not be bunched together, otherwise the station will not be able to cope. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded. For instance, if a particular station can only cope with at most two thirds of the cars passing along the line, the sequence must be built so that at most 2 cars in any window of 3 require that option. The problem has been shown to be NP-hard.

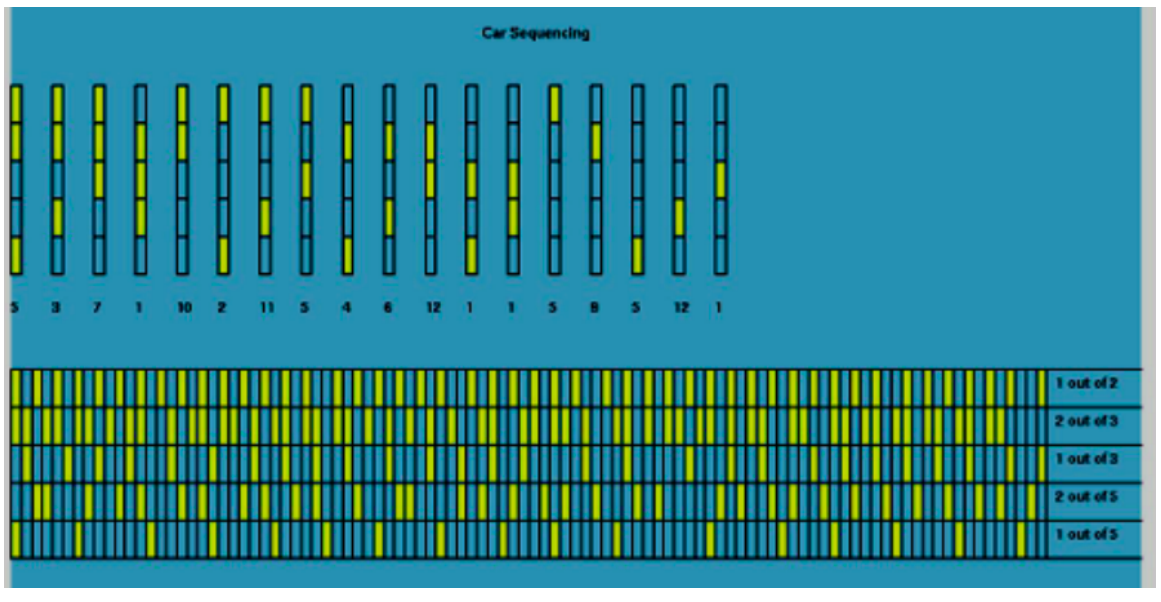
Informal description

Assembly line keeps on moving in front of the production units (imagine airport belts).

The production units are responsible for putting options on the car.(Eg: leather seats, sunroof).

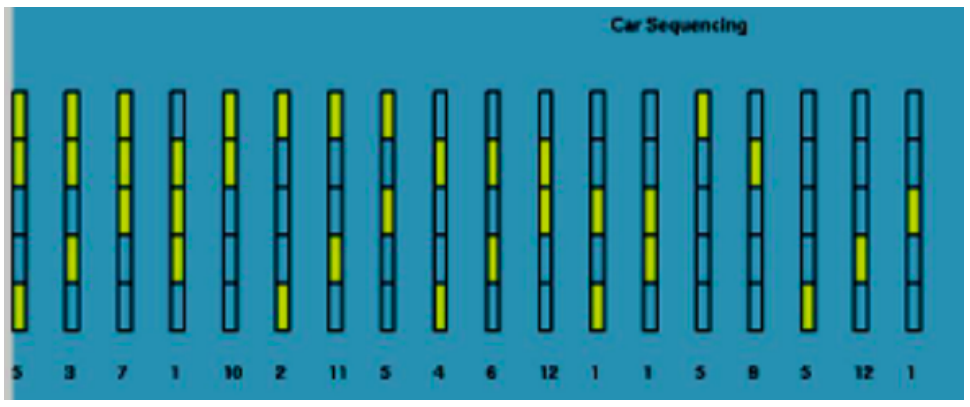
The cars are moving, so there is a capacity constraint that for example, says that : I can put this option(leather seats or sunroof) on atmost 'x' cars in a row of 'y' successive cars. Eg: I can put leather seats in atmax 2 cars among 5 successive cars. Else, the production unit won't have the time to put it on.

Aim: Sequence the cars such that all the capacity constraints are satisfied

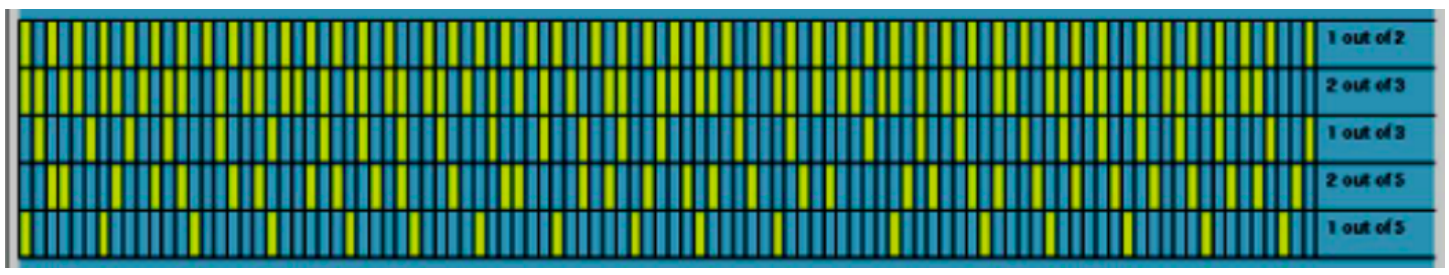


An attempt at a visualisation of the assembly line in a car factory

Trying to break the graphical representation into two parts:



In the above picture, there are 18 big rectangles. Each big rectangle represents a particular car type. Each big rectangle has been divided into 5 smaller rectangular parts: each part representing a particular feature(alloy wheels etc). If yellow→ that car type needs that feature; if blue, that car type does not need that feature. The number at the bottom of each big rectangle(car type) is the number of cars of that particular type which are present.



This is one particular snapshot of the assembly line. The number written on the right is a capacity constraint for a particular feature (Airbag, ABS etc) of the form **'x' out of 'y'**→ meaning that in all possible consecutive(successive) tuples of 'y' cars, only 'x' can demand for that particular option.

Defining a local move: swapping of any two slots on the assembly line (ie car $C1$ initially at slot $S1$ is placed at slot $S2$ inhabited by car $C2$ and vice versa)

Search strategy

- find a configuration that appears in violations
- swap that configuration with another configuration to minimize the number of violations

Discussing one particular snapshot

Slots	1	2	3	4	5	6	7	8	9	10	Demand
Class 1	■										1
Class 2		■									1
Class 3			■	■							2
Class 4					■	■					2
Class 5							■	■			2
Class 6									■	■	2

Initial configuration

In this picture, slot 1 has been allotted to car of class 1, slot 2 has been allotted to a car of class 2; slots 3,4 have been allotted to car of class 3 and so on.

Options	1	2	3	4	5	Demand
Class 1	yes		yes	yes		1
Class 2				yes		1
Class 3		yes			yes	2
Class 4		yes		yes		2
Class 5	yes		yes			2
Class 6	yes	yes				2
Capacity	1/2	2/3	1/3	2/5	1/5	

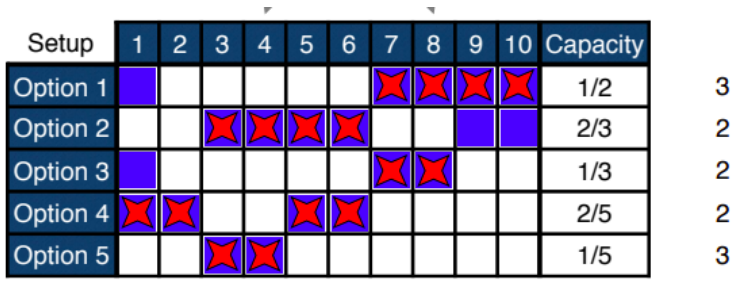
The above table has car classes 1 to 6 whereas options 1-5 represent the special service (ABS, airbags etc). The red text in demand column is the number of cars of type 'class i' of row 'i'. The red text at the bottom(where the row has been labelled as **capacity** is of the form x/y which can be interpreted as follows: in 'y' consecutive slots, atmost 'x' cars can demand for service mentioned in the respective column.

Setup	1	2	3	4	5	6	7	8	9	10	Capacity
Option 1	■						■	■	■	■	1/2
Option 2			■	■	■	■			■	■	2/3
Option 3	■						■	■			1/3
Option 4	■	■			■	■					2/5
Option 5			■	■							1/5

Initially, each of the cars have been allotted their own columns and if the cell (i,j) is blue, this means that car in slot 'j' demands service denoted by option 'i'.

The first step would be to check the feasibility of the above configuration and to check how many violations are present in it. A systematic way to do this would be to the following for each option: For an option **O** having capacity **x/y**, move a sliding window of length 'y' along the row belonging to option 'O' and determine the number of cars '**Z'** in the current window needing that option the number of blue cells in the sliding window). if 'Z' > 'x', then a violation occurs. After doing the above for each option, we get the following representation of the current configuration (Here, the **RED star** within the blue cell represents that the car in that particular slot is in one of the violations due to it demanding that particular service.

Setup	1	2	3	4	5	6	7	8	9	10	Capacity
Option 1	■						✦	✦	✦	✦	1/2
Option 2			✦	✦	✦	✦			■	■	2/3
Option 3	■						✦	✦			1/3
Option 4	✦	✦			✦	✦					2/5
Option 5			✦	✦							1/5



Setup	1	2	3	4	5	6	7	8	9	10	Capacity	
Option 1	■						✕	✕	✕	✕	1/2	3
Option 2			✕	✕	✕	✕			■	■	2/3	2
Option 3	■						✕	✕			1/3	2
Option 4	✕	✕			✕	✕					2/5	2
Option 5			✕	✕							1/5	3

So, we see option 1 appears in 3 violations, option 2 appears in 2 violations and so on. (Number of violations due to a particular option is present beside the right edge of the table).

Defining one particular neighbourhood of the problem

Configurations $'C_1'$ and $'C_2'$ are neighbours of each others if and only if both can be obtained from each other by swapping the cars in exactly one pair of slots.

The next step would be to choose a pair of columns(columns having cars of different types) and swap them together to make a try to reduce the number of constraint violations.

Here, we notice that instead of assignment to decision variables, we are making swaps. By designing our local move in his manner, we are making sure that the **demand constraint** (the total number of cars of a particular type 'x' in the input is equal to the total number of cars of that type in our configuration) hold implicitly.

The constraints can be divided into 2 classes:

- Hard constraints : always feasible, non-negotiable and any config must satisfy them. Eg:in magic square problem, the fact that all number entries must be distinct is a hard constraint
- Soft constraints: may be violated during search but must be satisfied in final solution. Eg:in magic square problem, the equality summation across rows,columns and diagonal is treated as a soft constraint

Hard Constraints: Constraints that are always satisfied during the search. :::spoiler Examples - Total no. of tasks scheduled - Total no. of cars of each type on the assembly line (demand constraint) - All numbers should be different in magic square assignment :::

Soft Constraints: Constraints that might be violated during the search but are necessary in the final output. :::spoiler Examples - No window of k time should have >m tasks - For each ‘option’ i.e. feature of car, no window of k slots should have >m cars requiring that option - Sum of numbers on each row, column, diagonal in magic square should be equal

MAGIC SQUARE

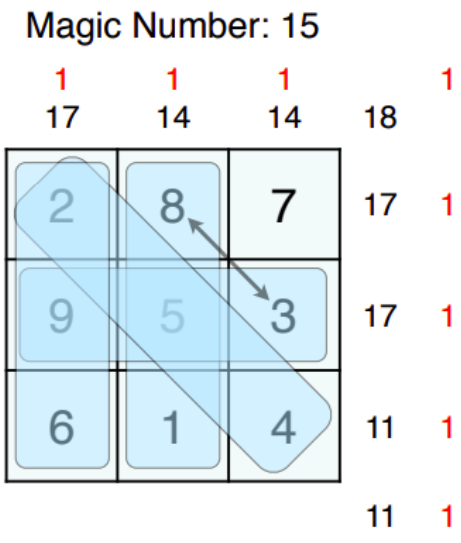
Statement: A magic square of order n is an arrangement of n^2 numbers, distinct integers, in a square, such that the n numbers in all rows, all columns, and both diagonals sum to the same constant (the magic number).

WHile solving, we classify the constraints as folowing:

Soft constraints: Sum of rows,columns, diagonals must be equal

Hard constraints: Numbers in cells cannot be repeated

Let us consider the case where the magic number is 15. Keeping **swapping values in 2 cells** as a local move ensures that the hard constraint is satisfied.



In the above initial configuration, the red font denotes the violation of constraint(truth value of 1) and initially all 8 (3+3+2) soft constraints are being violated.

The course of action now is to swap the values and to try to reduce violations.

Here is a concern which may arise in your minds:

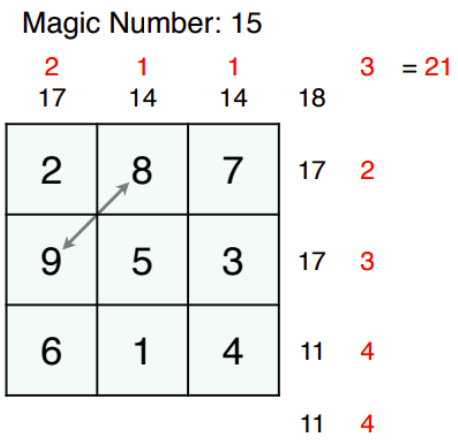
Even swapping may or may not look good although it may finally lead to a correct soln. How do I know whether the swap would be beneficial or not ? The initial incentive of reducing some violations may not lead up to the final goal : reduce all violations. If i just look at the 0/1 values, we may not have enough information to drive the search. For eg: if calculated sum across a row being 14 instead of 15 and the sum across a row being 1 instead of 15, these 2 scenarios represent different information and the next local move should depend on this information. In 1st instance (sum=14), making a minor increase in one of the cells should work whereas in the second case (sum=1), potentially lots of cells in the row across which the sum is being calculated must be swapped with bigger values.

This calls for a change in methodology of representing constraints:

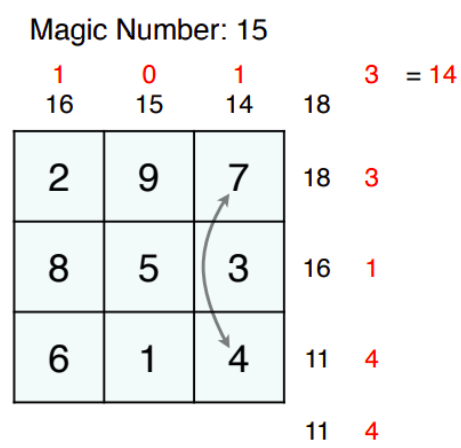
Instead of a strict 0/1 bound for constraint violation, we can have the constraint score to the degree to which a constraint has been satisfied or not. In this case, we can use **abs(expected-current)** as score of a violation/constraint.

- Swapping the value of two cells
  - the alldifferent constraint as a hard constraint
    - swaps are used in many permutation problems
  - the inequalities are the soft constraints
- What the violations?
  - 0/1 violations would be pretty useless
  - many moves would not change the violations
  - purely random walk
- For an equation l = r
  - use abs(l - r) as a measure of violations
  - drives the search much more effectively

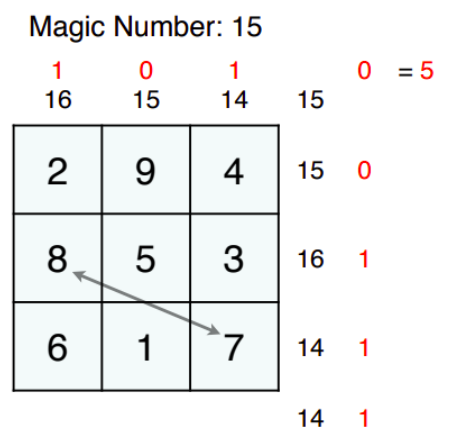
In below picture, overall violation score=21:



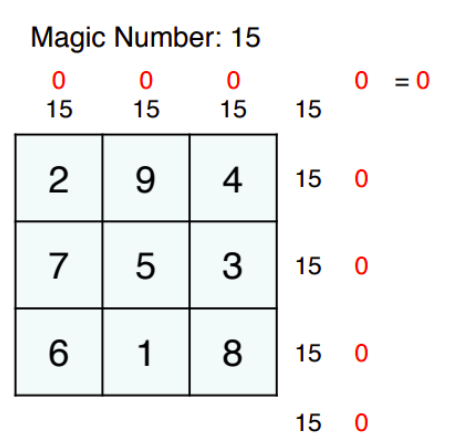
After 1 swap, overall violation score reduced to 14



After 2nd swap, overall violation score reduced to 5:



After 3rd swap, overall violation score reduced to 0:



Warehouse location (brief overview)

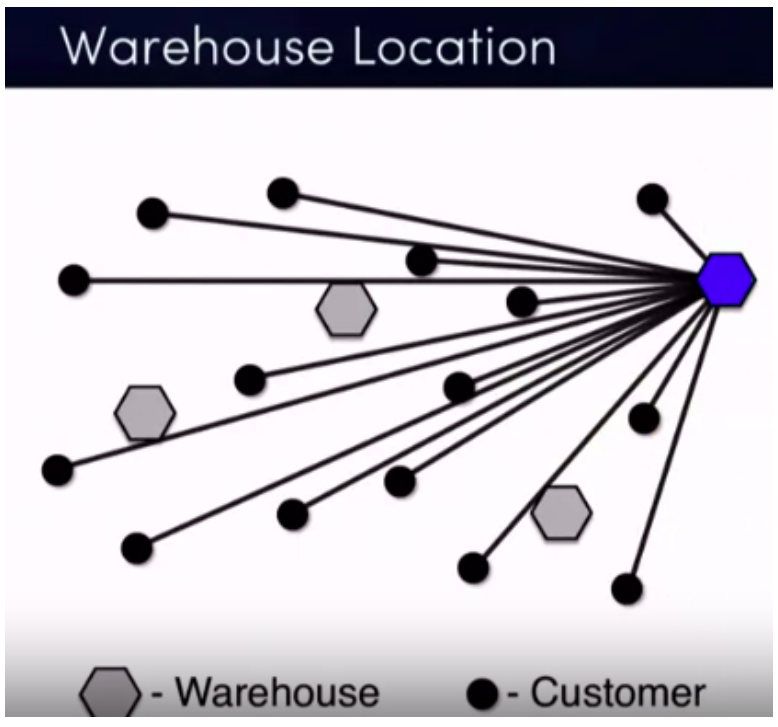
The study of facility location problems is concerned with the optimal placement of facilities to minimize transportation costs while considering factors like avoiding placing hazardous materials near housing, and competitors' facility.

Two main types of costs to be taken care of:

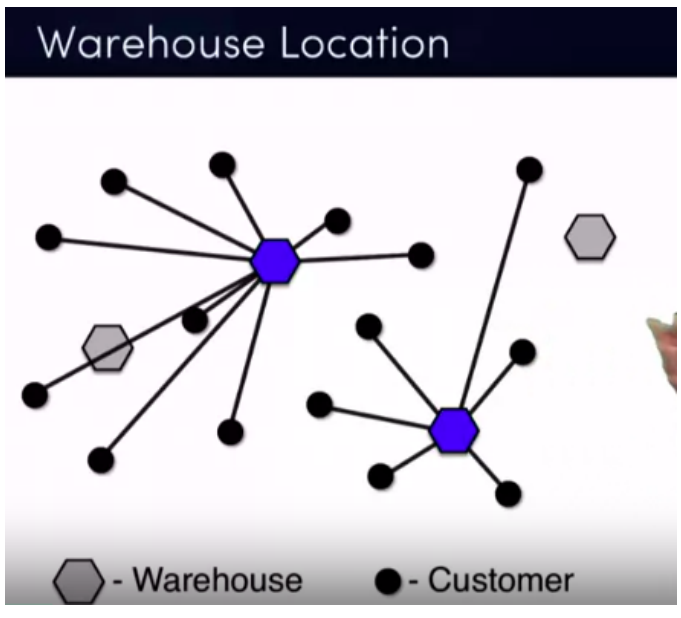
1. Cost of setting up the warehouse
2. Transportation cost to a customer from its nearest warehouse

GREY HEXAGONS: Possible locations

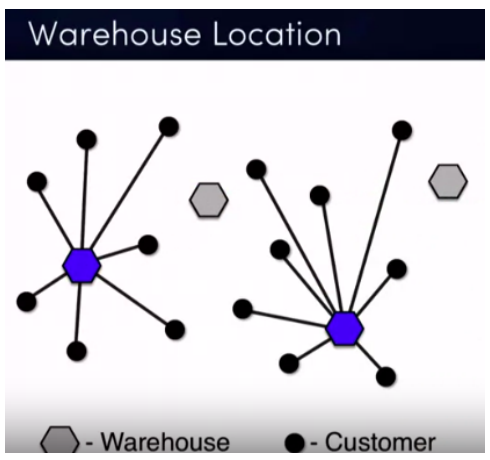
BLUE HEXAGONS: Locations chosen



Possibility 1

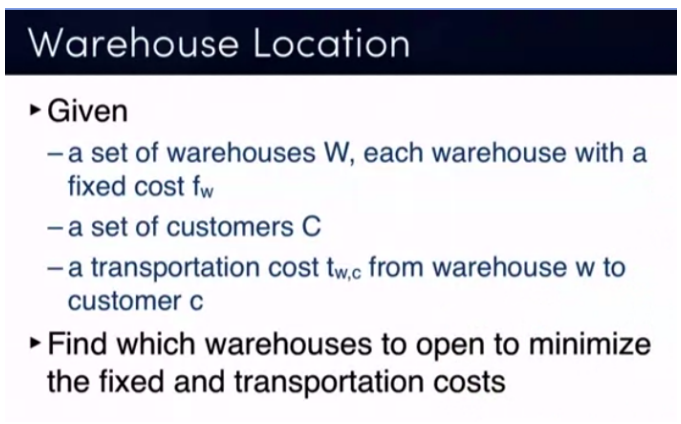


Possibility 2

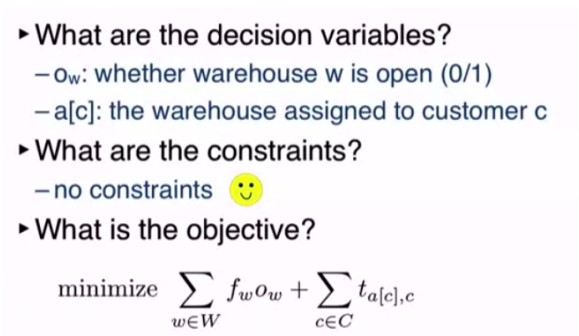


Possibility 3

The inputs given:



NOTE THAT THIS IS A objective function minimization problem rather than a constraint satisfiability problem.



- Key observation
  - once the warehouse locations have been chosen, the problem is easy
  - it suffices to assign a customer to the open warehouse minimizing its transportation cost
- What is the objective?

$$\text{minimize} \sum_{w \in W} f_w o_w + \sum_{c \in C} \min_{w \in W : o_w = 1} t_{w,c}$$

Deciding which configs (state) will be a reachable from a present state :

- Either opening or closing a particular warehouse
- Swapping the open/close value of a pair of warehouses

- Neighborhood
  - many possibilities
- Simplest neighborhood
  - open and close warehouses
  - that is, flip the value of o_w
- Union of neighborhoods
  - open and close a warehouse
  - swap two warehouses
    - close one and open the other

Optimality and feasibility

Solving feasibility and optimality together

There are many such problems with some constraint and a quantity to optimize.

Backdrop to graph coloring : Randomly generated Graph having about 250 vertices and the probability of having an edge between two vertices is 0.1%. Color this graph and also at the same time find the minimum number of colors needed to do the job.

Eg: Graph Coloring Constraint: No 2 edges should have same colored vertex

Shashwat's take

There are 3 approaches to such problems:

1. For a certain value of to be optimized quantity, find a feasible solution. Then improve the quantity slightly and switch things around to make the solution feasible.

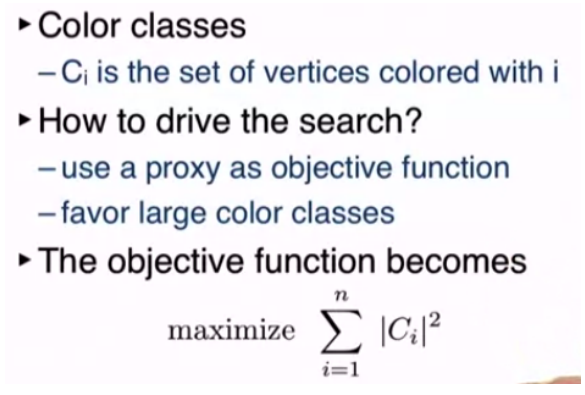
- Sequence of feasibility problems
  - find an initial solution with k colors
    - greedy algorithms
  - remove one color, say k.
    - reassign randomly all vertices colored with k with a color in the range 1..k-1
  - find a feasible solution with k-1 colors
  - repeat

How to make solution feasible with k-1 colors? Try to minimize constraint violations using local search

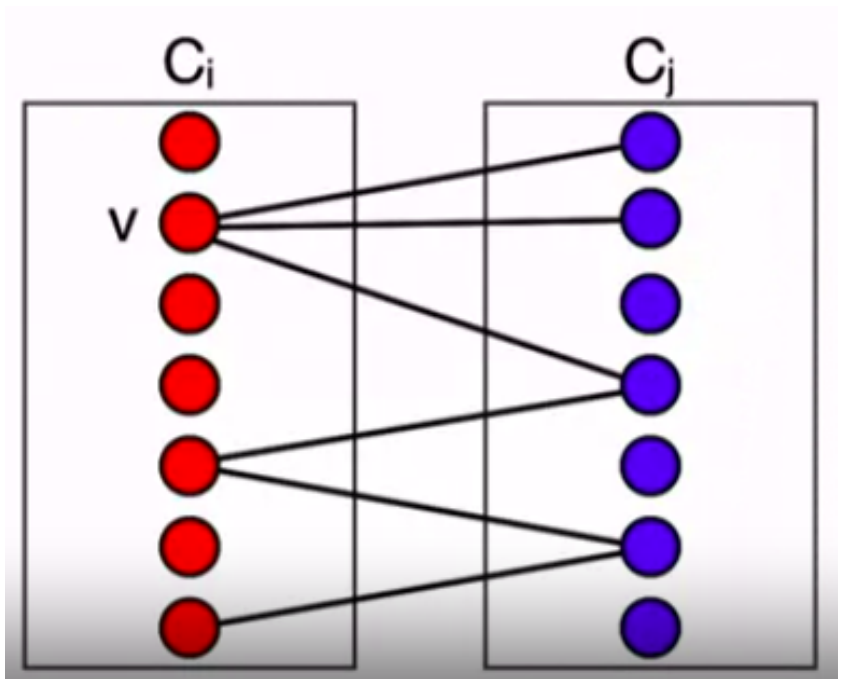
2. Always stay in feasible solution space, i.e. dont violate constraints at any stage. Then see if you can improve the optimization quantity using some changes such that you don’t violate the constraints. For this, you may want to use a different loss function that keeps the essence of the original optimization quantity but is more easy to evaluate ‘improvement’ in.

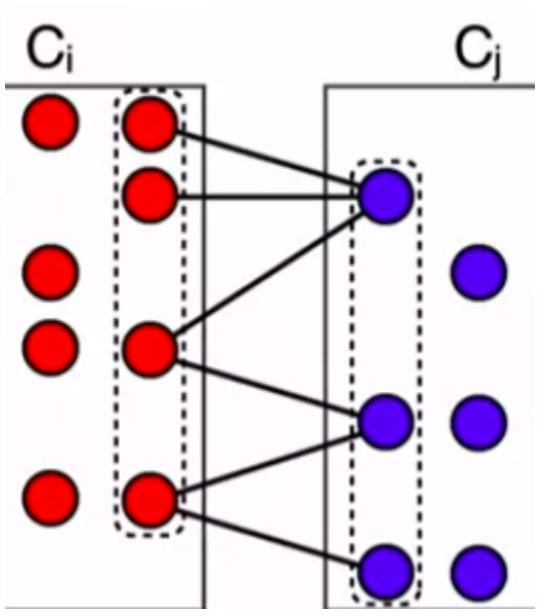
Graph coloring example

The original loss function is just number of colors. With a local move of 'change color of one vertex', you most probably won't improve the loss function. Therefore we need a loss function that conveys more verbose information. The loss function basically improves whenever we have larger color classes. So most color change moves would bring about a change in the loss function, and a change for the better if a node's color changes from i to j s.t. $C_i < C_j$ because we make progress towards removing the i^{th} color. Ofcourse this might lead to bad local minima as it's more of an approximation.



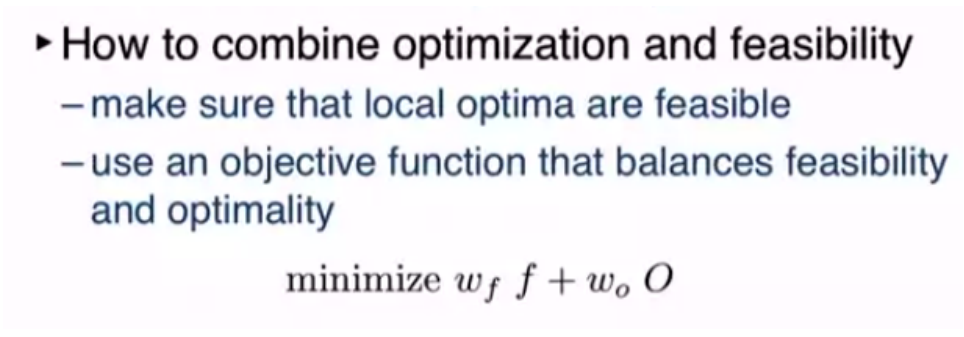
But we also have to ensure each move keeps the solution feasible. How to do that? Well, each move is not a single color change, but a sequence of color changes that lead to a feasible solution, called a **kemp chain**





Changing v from C_i to C_j would cause a match in colors with some adjacent vertices, so we switch those adjacent vertices to C_i and further on similarly. Sometimes (as in the image examples) it leads to an improvement in cost function (from 7-7 to 8-6) and these are the local search moves we make. So our neighbourhood consists of valid kemp-chain moves, not a single vertex color change now. :::

3. Do a mix of the above 2 strategies, i.e. bring improvements in either feasibility or optimization-quantity or both with each move.



Graph coloring example

Define B_i as set of bad-edges (same colored ends) of ith color. We want to minimize

$B_i\ (to\ 0)$ and maximize $|C_i|^2$ to minimize colors. How to combine them? Well we want to try that the new objective function ensures that our local minima are atleast feasible (even though we can't say anything about optimality).

Here, this can be ensured using the objective function $2|B_i||C_i| - |C_i|^2$.

. Proof:

- Consider a coloring $C_1,\ldots,C_k$
- Consider an additional color k+1
  - Select an edge in B_i and color one vertex with k+1
- How does the objective vary?
  - the left term decreases by

$$2|B_i||C_i| - 2(|B_i| - 1)(|C_i| - 1) = 2|B_i| + 2|C_i| - 2 \geq 2|C_i|$$

  - the right term increases by

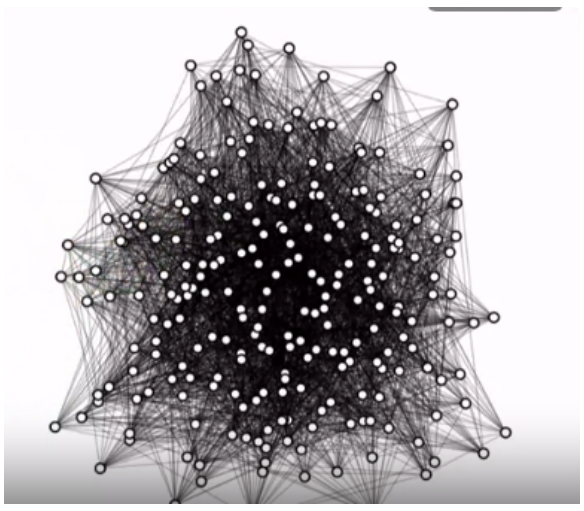
$$|C_i|^2 - ((|C_i| - 1)^2 + 1) = 2|C_i| - 2.$$

  - Overall, the objective decreases by at least 2

Thus the minima won't be a local minima if $|B_i| > 0$ for any i,, and thus the local minima are always feasible, though they are built towards by improving both feasibility and optimality iteratively.

Anmol's take

So, know this problem has two aspects:



Optimization: Reduce number of colors

Feasibility: Two adjacent vertices must be colored differently

Optimize: Minimize the no. of colors used

There is a tradeoff between feasibility and optimization.

Now, how can local search handle this problem:

Way 1: model problem as a sequence of feasibility problems; Try to find a feasible solution for n=250,n=249,2n=248 where n=number of colors required to color the graph

Way 2: Make this as an optimization problem only but in that case, the search space (alllowed configs) would only be the configs which satisfy all the constraints

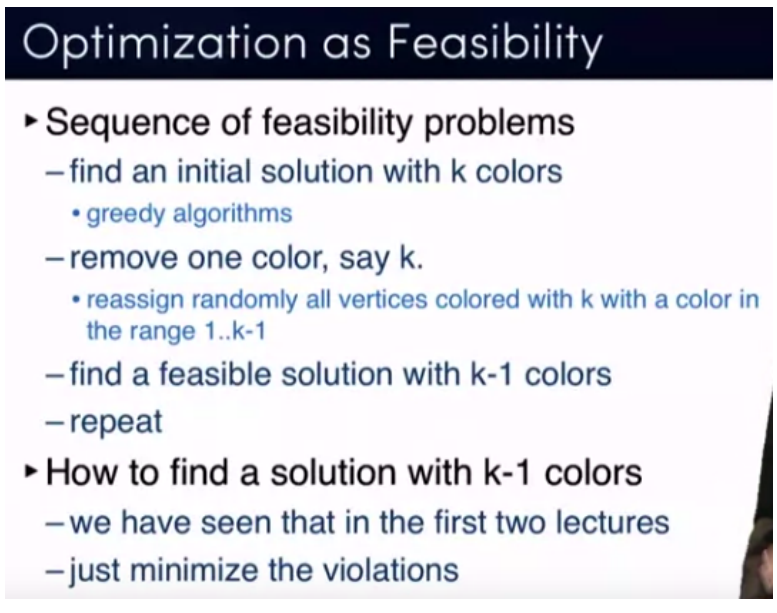
Way 3: Hybrid: Allow all sorts of configurations while also allowing for violations etc in between and still attempt to do the job in minimum number of colors

- Two aspects
  - optimization
    - reducing the number of colors
  - feasibility:
    - two adjacent vertices must be colored differently
- How to combine them in local search?
  - sequence of feasibility problems
  - staying in the space of solutions
  - considering feasible and infeasible configurations

Exploring way 1: Optimization as feasibility

So, basically we start with a known feasible soln with 250 colors in this case and next, we try to move from 250 to 249, 248 and so on until we fail. The only thing is that the failure detected would be via local search and so

we cannot guarantee that no solution exists for the optimization value goal (number of colors) for one which we failed. Try minimizing violations like you did in the n-queens problem



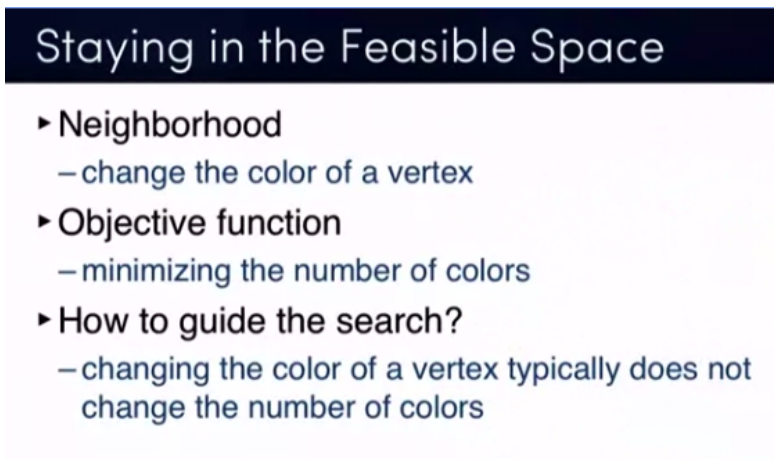
Exploring way 2: Staying in feasible space

If one does not like feasibility problems, one would instead try this option out.

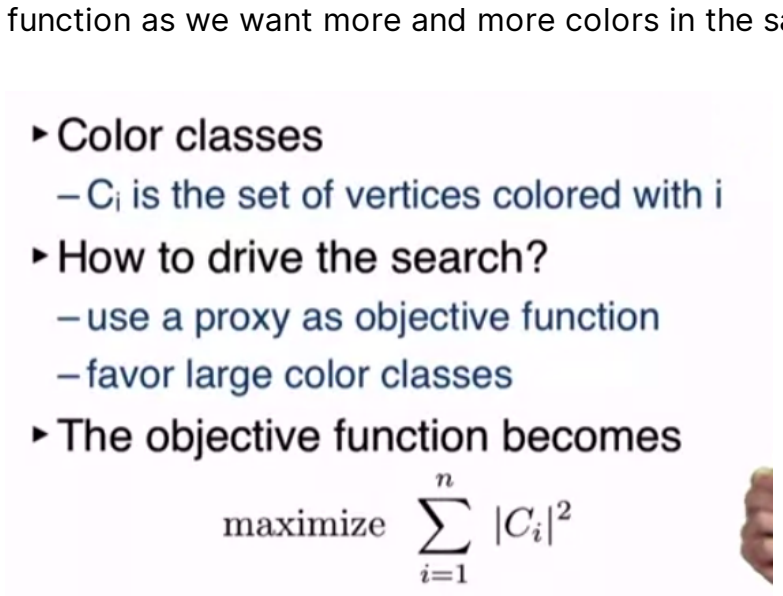
Our configs must be valid(satisfying all constraints) at each step. Only aim now would be now to minimize number of colors.

Objective : minimize number of colors

Challenge: If you change the color of a vertex, you are not likely to get any new info as in almost all cases, the total number of colors would still be the same. Eg: if 3-R,4-G,5-B, then if we pick any of the 12 vertices and change their color, the total number of colors used will still be 3 and hence, we won't be able to gain any new info. THis is similar to the problem we faced while dealing with the **magic square problem.**



Possible soln: Use an indirect optimization function based on the fact that $(a+b)^2$ is greater than a^2+b^2 and so it is better to MAXIMIZE this function as we want more and more colors in the same color set.

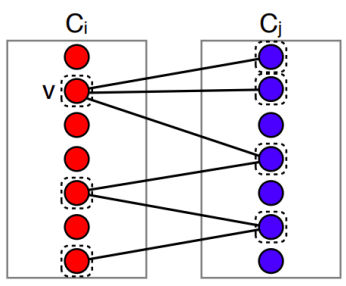


USE OF KEMP CHAIN

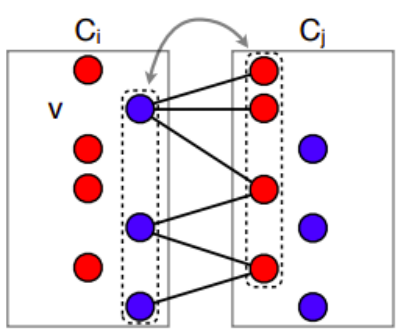
Recursive changing of colors so as to not break any violations (and the violations are solved by taking changing colors of other connected vertices).

Staying in the Feasible Space

- Richer neighborhoods
  - exploiting problem structure better
- Kemp Chains



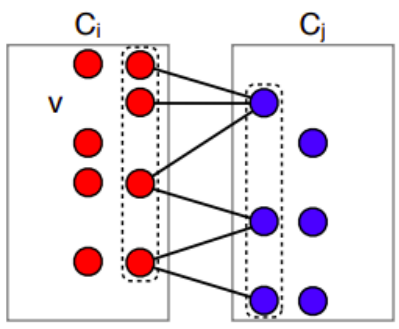
- Richer neighborhoods
  - exploiting problem structure better
- Kemp Chains



1

Staying in the Feasible Space

- Richer neighborhoods
  - exploiting problem structure better
- Kemp Chains



2

Exploring way 3: Dealing with both feasibility and optimization at the same time

So, essentially the aim in this case would be to design an objective function which penalizes a config both for violating constraints as well as for using large number of colors

Exploring both Feasible and Infeasible Colorings

- Explore both feasible and infeasible colorings
  - the search must focus on reducing the number of colors and on ensuring feasibility.
- How to combine optimization and feasibility
  - make sure that local optima are feasible
  - use an objective function that balances feasibility and optimality

$$\text{minimize } w_f\ f + w_o\ O$$

SO, defining bad edges:

The bad edges would penalize for edge between vertices of same color.

Exploring both Feasible and Infeasible Colorings

- Neighborhood
  - change the color of a vertex
- Bad edges
  - a bad edge is an edge whose adjacent vertices have the same color
  - B_i is the set of bad edges between vertices colored with i

The first part of objective function focuses on minimizing number of colors used. So, this idea/function can directly be borrowed from **Way 2**.

- Neighborhood
  - change the color of a vertex
- Decreasing the number of colors

$$\text{maximize} \sum_{i=1}^{n} |C_i|^2$$

- Removing violations

$$\text{minimize} \sum_{i=1}^{n} |B_i|$$

- How to combine them?

Proposed function is very clever due to its particular design. In general, for a generalized optimization function which would give weightage to satisfiability function and to the number of colors function. But if we reach a local minima, it is still possible that there are till colors violations (i.e. violations where two diff employees who are connected by an edge have the same color).

By designing the cost function intellectually, we make sure that whenever we reach a local minima, we would at least be sure that the vertices having same colors should not have a common edge) is satisfied.

The Combined Objective Function

- Neighborhood
  - change the color of a vertex
- Objective function

$$\text{minimize} \sum_{i=1}^{n} 2\,|B_i|\,|C_i| - \sum_{i=1}^{n} |C_i|^2$$

- Why?

local minima of this objective are legal colorings

Claim: local minima are legal colorings

Proof:

- Consider a coloring $C_1,...,C_k$
  - assume that B_i is not empty
  - we show that this coloring is not a local minimum
- Consider an additional color k+1
  - select an edge in B_i and color one of its vertices with k+1 (instead of i)
- Consider the objective

$$\text{minimize} \sum_{i=1}^{n} 2\,|B_i|\,|C_i| \;-\; \sum_{i=1}^{n} |C_i|^2$$

- How does it vary?

Local Minima are Legal Colorings

- Consider a coloring $C_1,...,C_k$
- Consider an additional color k+1
  - Select an edge in B_i and color one vertex with k+1
- How does the objective vary?
  - the left term decreases by

$$2|B_i||C_i| - 2(|B_i|-1)(|C_i|-1) = 2|B_i| + 2|C_i| - 2 \geq 2|C_i|$$

  - the right term increases by

$$|C_i|^2 - ((|C_i|-1)^2 + 1) = 2|C_i| - 2.$$

  - Overall, the objective decreases by at least 2

Complex neighborhoods, sports scheduling

- Input
  - n teams
  - a matrix d of distances between teams
- Output: a double round-robin schedule
  - atmost constraint: no more than three consecutive games at home or away
  - no repeat constraint: a game a @ b cannot be followed by a game b @ a
  - minimize travel distance

The travel distance is the sum of distances travelled by each team over the entire tournament.
So essentially we have to a fill a schedule (Sch) table of teams x rounds size, specifying the i^{th} teams plays the j^{th} round with team $Sch[i][j]$ and also add info. on which team is home and away in this matchup.

Intuition for first constraint

Had the first constraint not existed, it might have been optimal to make teams play a lot of home matches together, and then go out on a tour of away games. But this gets boring for fans during the away part of the season and also causes teams with an early home phase to take a lead in the season only to be caught up later, i.e. scoreboard becomes more subtle/complex, which is undesirable.

The interesting part of this problem is defining a neighbourhood for a particular assignment. Something as simple as swap 2 rounds for a particular team might not work well alone (because it also propagates changes elsewhere in the table for the opponent teams). The current best solution to this problem constructs the following moves to define a complex neighbourhood:

1. **Swap homes**: For given pair of teams A and B, swap which game is at A's home. It doesn't affect much, except the first constraint and possibly reducing travel distance.
2. **Swap rounds**: The exact round number for matchups is irrelevant, so you could take 2 columns (rounds) and swap them. This is a big move, i.e. radical change is possible.
3. **Swap teams**: Take the sequence of matches for two teams (two rows), and swap them. But now the opponents schedule also gets affected, so changes have to be made to maintain the integrity ($Sch[i][j] =$

$Sch[Sch[i][j]][j]$) of the table acc to new schedule. Keep their games with each other as it is. Another radical change.

4. **Swap partial rounds**: Pick 2 rounds, and instead of swapping them entirely, exchange around the schedule for teams within these 2 rounds while maintaining integrity.
5. **Swap partial teams**: Instead of swapping the whole row, swap a subset of round matches of 2 picked teams. Now integrity would be violated in the table elsewhere, so fix that too.

Is 2-opt connected or not ?

In 2 opt, imagine the tour as a sequence of vertex numbers. WHat 2-opt essentially does is that it reverses a particular subsequence while moving from one configuration to another.

Question : Is TPP (travelling tournament)s neighborhood connected ?

LS 7 - formalization, heuristics, meta-heuristics introduction

Defining the termHeuristics

A heuristic algorithm is one that is designed to solve a problem in a faster and more efficient fashion than traditional methods by sacrificing optimality, accuracy, precision, or completeness for speed. Heuristic algorithms often times used to solve NP-complete problems, a class of decision problems. In these problems, there is no known efficient way to find a solution quickly and accurately although solutions can be verified when given. Heuristics can produce a solution individually or be used to provide a good baseline and are supplemented with optimization algorithms. Heuristic algorithms are most often employed when approximate solutions are sufficient and exact solutions are necessarily computationally expensive.

Trade-off [edit]

The trade-off criteria for deciding whether to use a heuristic for solving a given problem include the following:

- *Optimality:* When several solutions exist for a given problem, does the heuristic guarantee that the best solution will be found? Is it actually necessary to find the best solution?
- *Completeness:* When several solutions exist for a given problem, can the heuristic find them all? Do we actually need all solutions? Many heuristics are only meant to find one solution.
- *Accuracy and precision:* Can the heuristic provide a confidence interval for the purported solution? Is the error bar on the solution unreasonably large?
- *Execution time*: Is this the best known heuristic for solving this type of problem? Some heuristics converge faster than others. Some heuristics are only marginally quicker than classic methods.

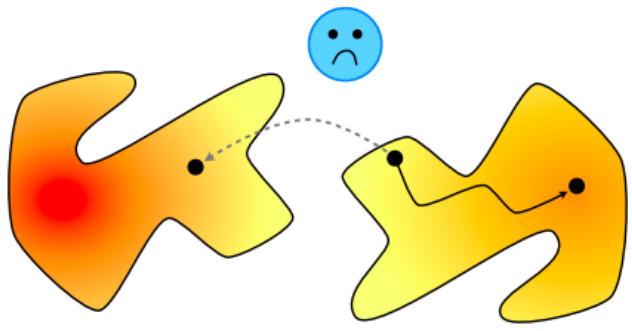
In some cases, it may be difficult to decide whether the solution found by the heuristic is good enough, because the theory underlying heuristics is not very elaborate.

For our purposes, the heuristics we use have a compromise in optimality in the sense that we aren’t guaranteed to find a global minima(or maxima) but may get trapped in a local minima(or maxima).

LS 8 - iterated location search, metropolis heuristic, simulated annealing, tabu search intuition

There are 2 main ways you can end up at non-global (henceforth called bad) optima:

1. You move into a bad optima and there's no way to move out
2. You start in a region which is disconnected from the global optima, so you can never reach it and end at a bad optima.



We define a 'connected' neighbourhood as:

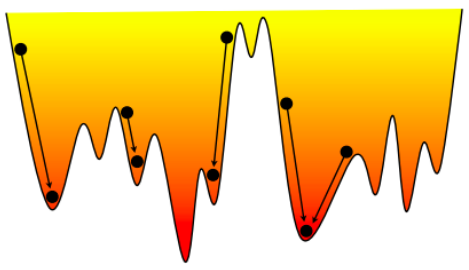
- A neighborhood N is connected if, from every configuration S, some optimal solution O can be reached by a sequence of moves, i.e.,

$$S = S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots. \rightarrow S_n = O$$

where

$$S_i \in N(S_{i-1}).$$

Connectivity doesn't guarantee optimality as in your greedy search strategy you might end up at a local minima. But it is essential for there to be some hope for reaching the global minima.



To prove connectivity, you assume the optimal solution and prove that there is a sequence of moves within your defined neighbourhood to construct the optimal solution from your starting configuration

▼ **Graph coloring example**

- Neighborhood
  - Change the color of a node
- The neighborhood is connected
  - simple algorithm
  - S_n is the color of node n in configuration S
  - O is the optimal configuration

```
S := some configuration
for each node n
  if Sn != On
    Sn := On
```

▼ Car Scheduling - Swap Neighbourhood

Swapping can take you from any permutation to the optimal one.

```
S := some configuration
for(int i = 1; i <= n; i++)
   if (Si != Oi)
      let Sj = Oi (j > i)
          Si <-> Sj
```

▼ Travelling Salesman - 2-opt

The neighbourhood defined by a 2-opt move (pick 2 non-connected edges, replace them with diff. 2 edges among the 4 nodes) is connected. This is because travelling salesman solution can be considered an array if we fix arbitrary node as the starting point. Each 2-opt move (assume perfect graph) corresponds to reversing exactly one contigous subarray of size >=2. Thus we can achieve any optimal permutation we want from any starting configuration.

```
T := some tour
for(int i = 1; i <= n; i++)
   if (Ti != Oi)
     find Si,...,Si+k such that Si+k = Oi
     S := (S1,...,Si-1,Si+k,Si+k-1,...,Si,Si+k+1,...,Sn)
```

Open question left by PvH: Is travelling tournament problem 5-step neighbourhood connected?!

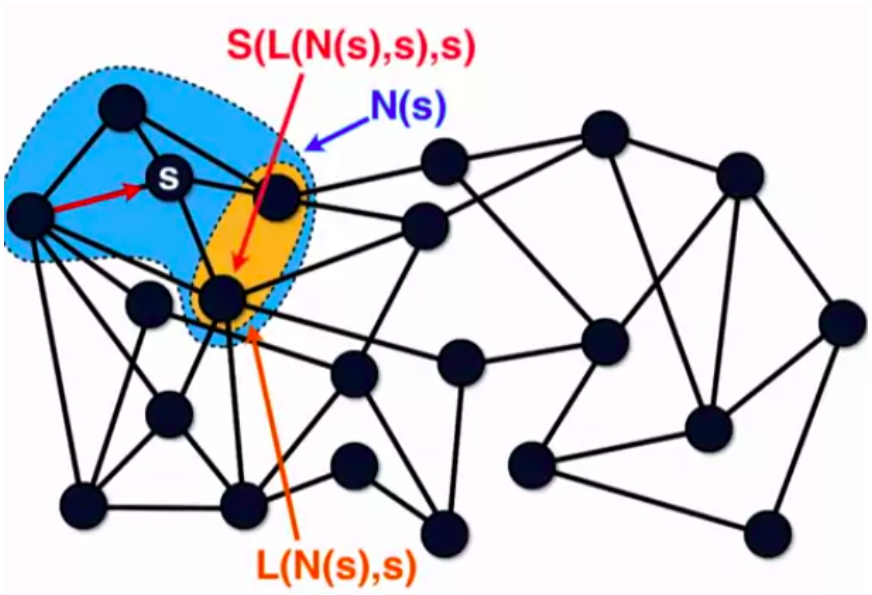
Local Search Implementation

a BRIEF OVERVIEW OF THE content covered till now and some notational stuff:

- **States:** either solutions or configurations
- **Moving from state s to one of its neighbours:** Here for a state s, $N(s)$ is defined to be the neighbourhood of a state s
- Some neighbours are legal and some are not: Let $L(N(s), s)$ be the set of legal neighbours
- **Selection function** (selecting one the the legal neighbours): $S(L(N(s), s))$
- **Objective function:** minimizing a function $f(s)$ where s is used to refer to a state

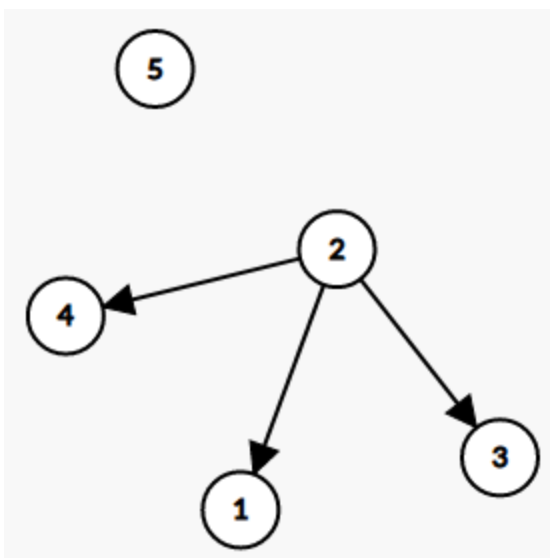
```
1.  function LocalSearch(f, N, L, S) {
2.      s := generateInitialSolution();
3.      s* := s;
4.      for k := 1 to MaxTrials do
5.          if satisfiable(s) ∧ f(s) < f(s*) then
6.              s* := s;
7.          s := S(L(N(s), s), s);
8.      return s*;
9.  }
```

f is the objective function, N is neighbourhood function, L is legal neighbourhood from given neighbourhood function and and S is the next-state selection from legal neighbourhood function. s is the current state. Algorithm basically says that start with initial solution, maintain $s*$ which is the best seen state till now. Then do some number of trials in which a new state is moved to by selecting it from the set of legal neighbourhoods which are selected from the set of neighbourhoods.



NOTE: 'Legal' here is not supposed to be interpreted as the allowed states to be only those states which do not violate any constraints. Rather, how we choose to define criteria for being a legal neighbour is upto us.

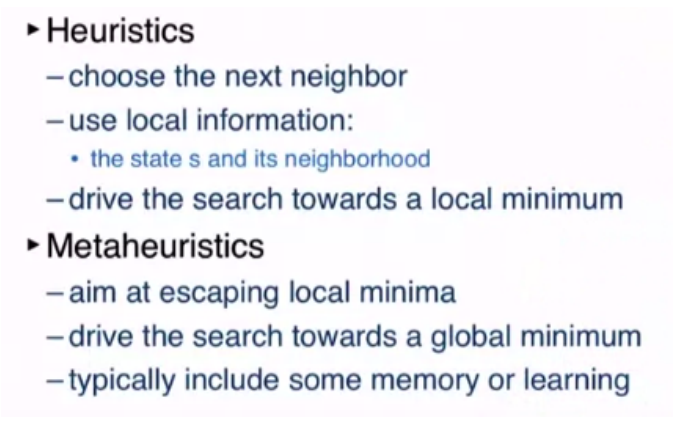
Eg: If we are currently at state 2 and $f(2) = 2$.
Let $f(1) = 1, f(3) = 3$ and $f(4) = 2, f(5) = 5$.
Now, $N(2) = 1, 3, 4$



Now, the vertices in $L(N(s), s)$ as per criteria would be as follows:

- Local improvement: $L(N(s), s) = \{1\}$
- No degradation: $L(N(s), s) = \{1, 4\}$
- Potential degradation: $L(N(s), s) = \{1, 3, 4\}$

Heuristics: Choose the next neighbour, driving the search towards a local minimum.
Metaheuristics: Try to escape a local minima, driving the search towards a global minimum. Typically include some memory or learning.



For example legal neighbourhood function might be 'all states that reduce f from the current state', reduce might be relaxed to 'dont degrade' or even 'select all ''(identity function). The selection function may be 'best neighbour', i.e. choose the one which gives best objective function. It may also be 'first neighbour', i.e. the first one found to be legal. Lastly something mixed can be done, called 'multi-stage selection' where we first select one part of the neighbourhood and then choose the best among these.

Defining which reachable states from $N(S)$ do we want to include in the $L(N(s), s)$ i.e. set of legal neighbours

One common way to select the set of legal neighbours would be to judge each neighbour on the value of the objective function for that particular neighbour. Note: Here we are assuming that the optimisation function needs to be minimised. **Neighbours can be classified into 3 types: based on how they perform in comparison to the current state.**

1. **Local improvement:** In this case, $L(N, s)$ consists of those neighbours in N(s) where the value of objective function is less than the value of the objective function in the current state

   Clearly,this can be represented as follows:

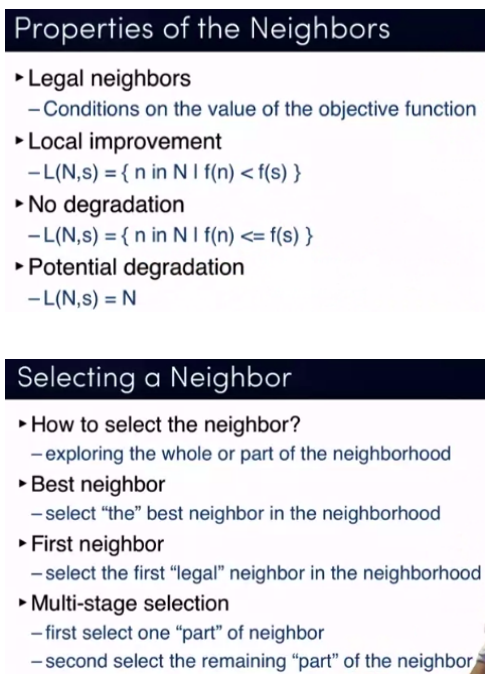
   $L(N, s)$ = {n in N | $f(n)$) < f(s) }
2. **No degradation :** In this case, $L(N, s)$ consists of those neighbours in N(s) where the value of objective function is less than or equal to the value of the objective function in the current state

   Clearly,this can be represented as follows:

   $L(N, s)$ = {n in N | $f(n)$) ≤ f(s) }
3. **Potential degradation:** In this case, $L(N, s)$ consists of all neighbours in N(s) irrespective of value of objective function

   Clearly,this can be represented as follows:

   $L(N, s)$ = N



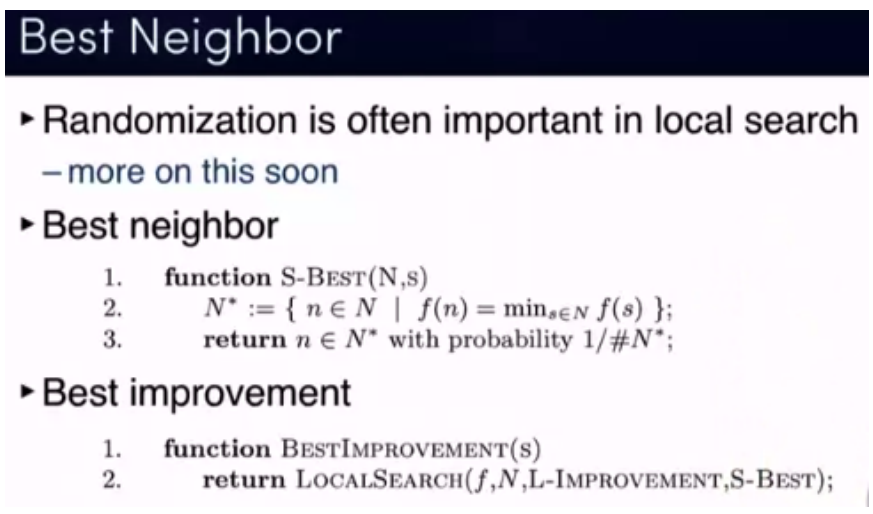
UNDERSTANDING OF BEST NEIGHBOUR VS BEST IMPROVEMENT:

> ⚙ Currently,you may be guessing where **best neighbour** can be applied except for passing it as an argument to the **best improvement** algorithm. Keep reading : Simulated annealing will come to your rescue.

In Best neighbour, there is no legality criteria. Let's say that I am on state 2 currently. And all neighbours of 2 ie in the set N(2) have value of optimisation function worse than 2. Even then, despite

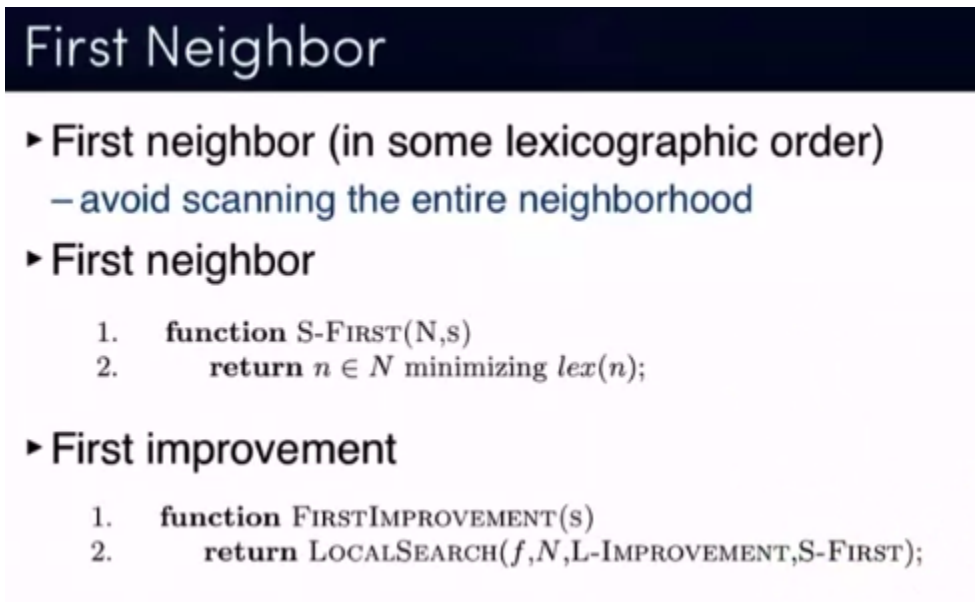
2 having a more favourable optimisation function value, I still choose one of the neighbours which is least worse.

In Best improvement, there is a legality condition as described by the 'local improvement'. So, in case no neighbour has a more favourable optimisation function value than the current state, then set of legal possible neighbours would be empty and hence, we would stay on the current state.

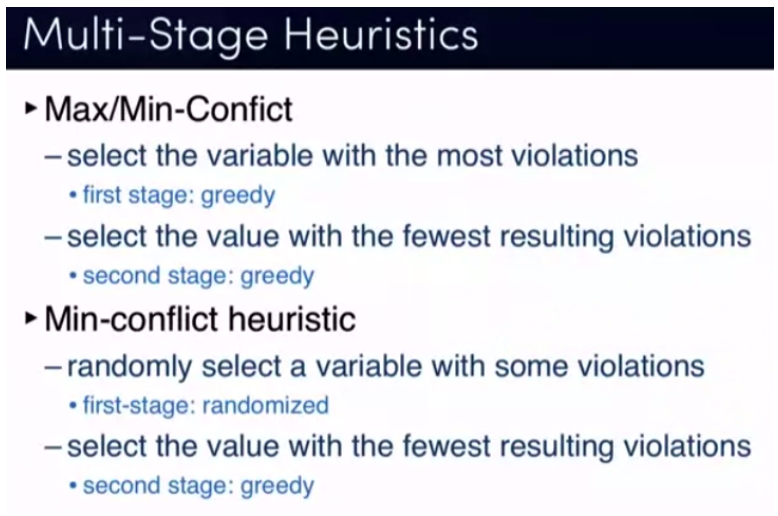


First Neighbour

Basically, **first improvement** algorithm is quite obvious from the name itself. Instead of visiting all legal neighbours and choosing the one in which we get the best improvement (if many such vertices exist, we choose one among them randomly); while traversing through the set of all neighbours, we choose the first vertex which satisfies the legality criteria.



Multi-stage heuristics



So, basically for the N-queens problem, we can explore the neighbourhood in 3 potential ways:

▼ **The 3 ways for N QUEENS PROBLEM**

Way 1 : Max/min conflict → So, we essentially decide the selection of the next neighbour in 2 stages. First, we choose the queen Q' which is occurring in maximum number of violations and then we choose the new position of queen Q' (after iterating through all positions this selected queen can reach by one just one valid chess move) which helps to get rid of maximum number of currently existing violations.

Time complexity : O(N) for deciding which queen to select and then O(N) for selecting which move that queen should make $\equiv O(N)$

Way 2: Min-conflict heuristic → Here, we try to select the queen Q' randomly and then take $O(N)$ to select the best move for this queen Q'. Overall complexity: $O(N)$

Way 3: BRUTE FORCE → Explore each and every move by each queen and see which move reduces maximum number of currently existing violations. Overall Complexity : $O(N^2)$

All the 3 ways are based on a greedy approach and hence, it seems better to not waste time in an $O(N^2)$ algorithm since it does not guarantee optimality. By using one of ways 1 or 2, we may use the time we saved (as compared to way 3) to explore configurations.

Multistage heuristics in Car Scheduling Problem

⚙️ We had initially defined a state S' to be in neighbourhood of state S if S' can be reached from S by swapping the positions of 2 cars of different types (with a different set of demands).

Setup	1	2	3	4	5	6	7	8	9	10	Capacity	
Option 1	■						✕	✕	✕	✕	1/2	3
Option 2			✕	✕	✕	✕			■	■	2/3	2
Option 3	■						✕	✕			1/3	2
Option 4	✕	✕			✕	✕					2/5	2
Option 5			✕	✕							1/5	3

Way 1: Quadratic neighbourhood

Here we try to swap each possible pairs of cars (where the cars have a different set of requirements) and check if this offers us the most improvement. This will lead to $O(N^2)$ possible pairings.

Way 2: Multistage neighbourhood

Alternatively, we can select a car C which is present in some violations (denoted in diagram by red) and try all possible swaps which include car C and choose the best possible move from here. This will lead to $O(N)$ computations.

📌 Way 1 offers the best move but way 2 offers a very good move in very less amount of time. So, again there is a trade-off between the time of the move and the quality of the move.

Examples of multi-stage heuristics

Max/Min Conflict: As stated earlier in the N-queens example, this means to select the variable with most violations, assign it the value that leads to least violations.

Min Conflict: Randomly select a variable which has violations, assign it the value that leads to least violations.

Essentially, choosing a multi-stage heuristic involves a tradeoff between the decrease in size of the part selected leading to possibly worse outcomes, but taking lesser time. Intuitively though, if you have a really large neighbourhood, it makes sense to choose a small part of it and then pick the best, and you can always find your way back during the local search as you can now run it for more iterations.

Random Walks

Choose a random neighbour, see if it improves performance, pick it if it does. Most effective when the neighbourhood is really large and ideas on how to logically reduce it to something smaller are not clear. Example: Travelling tournament problem. In TTP, some local moves were going to take as bad as $O(N^3)$ complexity and hence, using random walks experimentally gave better results in the past. In-fact, this is used by the best result for which the 5 neighbourhoods were described earlier.

Random Walks

- Randomization
  - select a neighbor at random
- Random improvement

```
1.  function S-RandomImprovement(N,S)
2.      select n ∈ N with probability 1/#N;
3.      if f(n) < f(s) then
4.          return n;
5.      else
6.          return s;
```

- Random improvement search

```
1.  function RandomImprovement(S)
2.      return LocalSearch(f,N,L-All,S-RandomImprovement);
```

Iterated Local Search

Re-start local search again and again with either a different initial configuration (might be chosen with some defined logic or randomly) or some randomization inside the algorithm, choose the best state obtained in all these iterations as the final solution.

Metrapolis Metaheuristic

If a move improves the objective function, take it. If it degrades it, take it with some probability. This probability gets lower the more the move lowers the objective function.

- inspired by statistical physics
- How is the probability chosen?
- t is a temperature
- Δ is the difference f(n) - f(s)
- a degrading move is accepted with probability,

$$\exp(\frac{-\Delta}{t})$$

So if t is very large, the probability converges to 1, so we almost always take the move and it tends to a random-walk. If t is very small, the probability converges to 0, so we rarely take the move and it tends to a greedy search. So we have to make a compromise between accepting degrading moves and moving towards only better solutions.

Simulated Annealing

Start with a very high temperature, essentially a random-walkish approach. Progressively cool the temperature, becoming more and more greedy (towards a normal local improvement search). A reasonably fast updatetemperature example is $t_k = \alpha*t_{k-1}$

```
function SimulatedAnnealing(f, N) {
    s := generateInitialSolution();
    t_1 := initTemperature(s);
    s* := s;
    for k := 1 to MaxSearches do
        s := LocalSearch(f,N,L-All,S-Metropolis[t_k],s);
        if f(s) < f(s*) then
            s* := s;
        t_{k+1} := updateTemperature(s,t_k);
    return s*;
}
```

For an extremely (unrealizably) slow search with cooling speed tending to 0, it is proven to converge to the global optimum [as it basically becomes like random-walk] But even in practice with a reasonably fast schedule also it does well in problems like TTP.

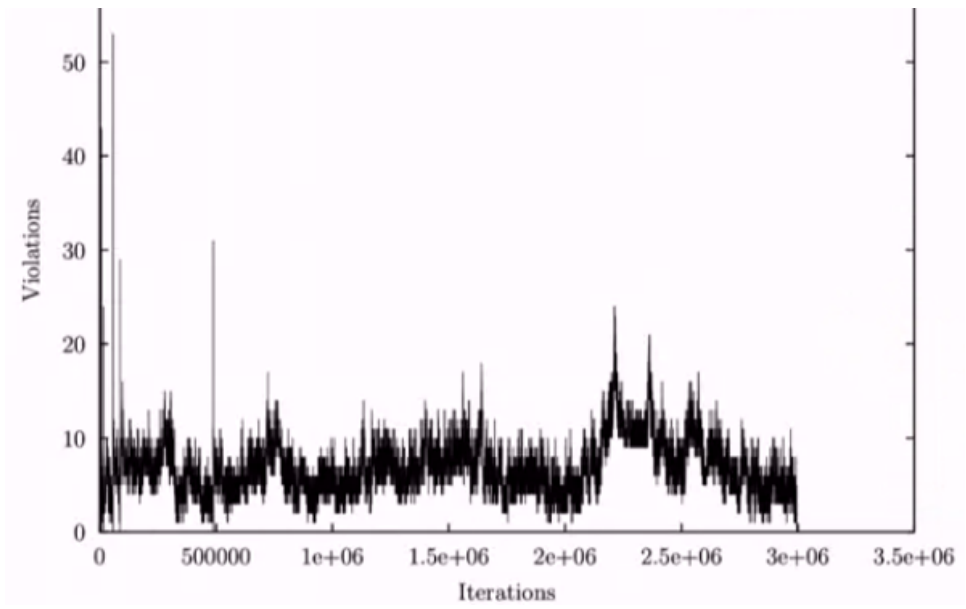
This can be combined with various techniques like re-starts (iterated) and re-heats (increase the temperature periodically so that you have spurts of 'breadth' and 'depth' in the search-space). Can also take insights from tabu-search.

Tabu Search

We maintain a list of 'tabu' states which we don't want to visit again. Intuitively, this can ensure our meta-heuristic won't make us back-track, and instead will force us to find new states/regions even if they are worse than some previously seen states in terms of the objective function. So,basically the idea of tabu search is once you have explored a particular state, you don’t want to go back to that particular state ie you mark it as TABU

Now, it's not necessary to mark every visited state as tabu, tabu selection function can also be programmer-decided.

Graph coloring tabu-search performance graph



Y-axis is number of colors and X-axis is iterations.

Metaheuristics

- Many others
  - variable neighborhood search
  - guided local search
  - ant-colony optimization
  - hybrid evolutionary algorithms
  - scatter search
  - reactive search
  - ...

Tabu-list selection

Timewise, tabu search is **expensive** as knowing whether a node has been already visited **would need iterating through all the already visited states and checking for equality.**

It may not even make sense to maintain the list of all visited states because each configuration is large to maintain in memory and we will have thousands/millions of configurations over time. We can use something like **short-term memory, where we only maintain the last X states, where X can change dynamically** (eg: higher X if taking a degrading move and lower if improving move). But even then it can be quite costly.

So we may want to store an "abstraction" of the tabu list, instead of explicitly storing disallowed states. Instead of storing states in the tabu-list, **we can store transitions.** The idea is not to make transitions that essentially undo recent translations, because not a lot of 'context' would have changed.

Example/Implementation of Car assembly

If transition is swapping to places, don't swap a recently swapped pair back

For a user-defined Tabu, maintaining a tabu list consisting of all visited states should enable us to prevent slots (i, j) which have been recently swapped from being swapped again, for at least some number of future iterations..

- How to implement this
  - keep an iteration counter it
  - keep a data structure tabu[i,j] which stores the next iteration when pair (i,j) can be swapped
    - an iteration number
    - not legal to swap this pair before
  - assume that the tabu list of size L
- When is a move (i,j) tabu?
  - when tabu[i,j] >= it
- What happens when applying a move?
  - tabu[i,j] becomes it + L

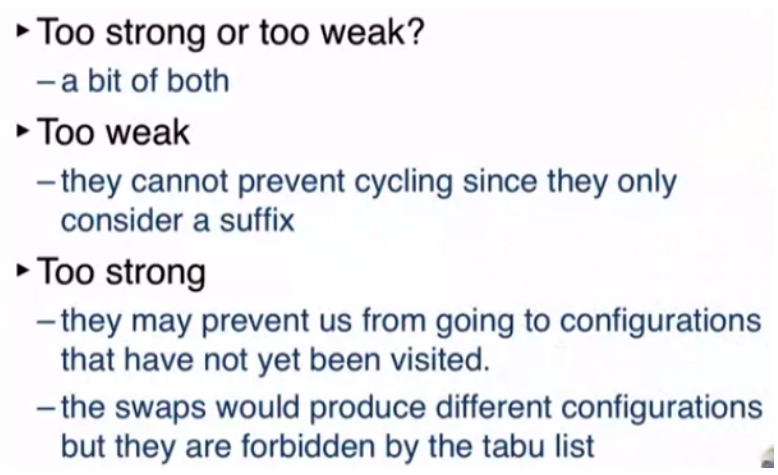
Illustration: Car Sequencing

- Tabu search: quadratic neighborhood

```
s := some initial configuration;
s* := s;
it := 0;
for(int i = 1; i <= n; i++)
  for(int j = 1; j <= n; j++)
     tabu[i,j] := 0;
while (violations(s) > 0) {
  N := { <i,j> | violations(i) > 0 & i != j };
  NotTabu := { <i,j> in N | tabu[i,j] <= it};
  if (|NotTabu| > 0)
    <i,j> := argmin(<i,j> in NotTabu) f(swap(s,i,j));
    s = swap(s,i,j);
    tabu[i,j] := it + L;
    tabu[j,i] := it + L;
 if (f(s) < f(s*))
    s* := s;
  it++;
}
return s*;
```

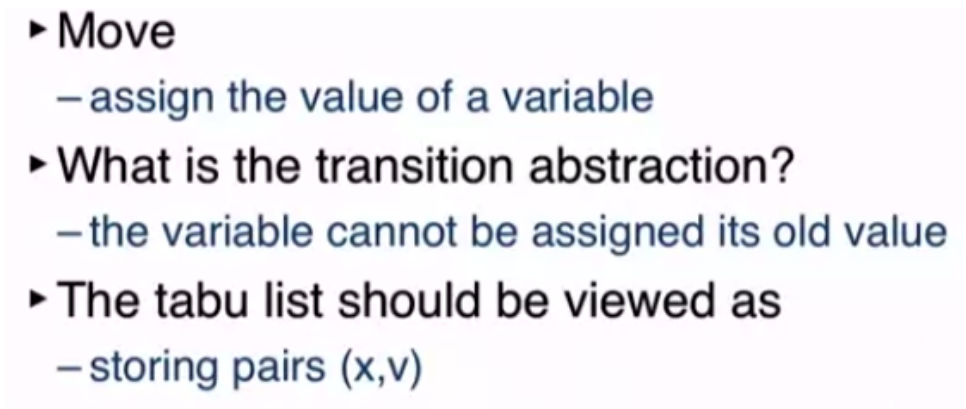
Reduce violations by making swaps that are not tabu. Red-lines represent tabu-search specific code. Notice that even if all moves are tabu right now, in the future the iterations will increase so some moves will start becoming available. In practice, moves are not implemented before the cost is calculated unlike the code example.

Calculating costs before making moves is called 'differentiation'.



Can also then be combined with multi-stage heuristic so instead of quadratic we can choose linear neighbourhood (for a particular car, find which car to swap it with).

Tabu list - Queen's problem example

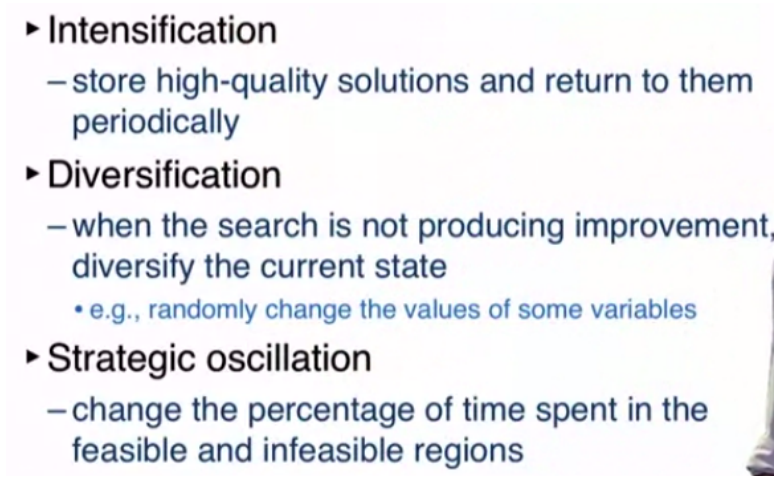


:::

Sometimes you may take a coarser tabu criteria, that makes the search more diversified. Like in the queens problem, don't allow the same queen to be assigned a new value for a certain number of moves.

Aspiration Criterion in Tabu Search: If a move not being allowed by tabu is too good, we might want to take it. We can set criterion for this such as if it's more than a certain magnitude improvement over the objective function.

Some more techniques:



In strategic oscillation if we are spending too much time in a local minima, we might increase the weight of the objective function and decrease weight of the constraints to come out. On the other hand, if we're violating too many constraints we may do the opposite. So the weights oscillate and take us to-and-fro from feasible region to infeasible region. These can also be used in simulated annealing, tabu-search or normal local search etc.

- If all possible local moves are tabue'd in a rare case, we would have to wait till the counter is incremented enough for a move to be not Tabu.
- Tabu search strategy does not fully exclude the possibility of the same configurations being explored again and again as we may visited an already visited state after its punishment time in the tabu list is over. Hence, there is still chance we will keep going in cycles.
- In cases where moves are tabued instead of previous configurations, the Tabu list restriction can prevent us from visiting a promising unvisited config C just because the local move we need to get to C is in the tabu list currently.
- To reduce the probability of us not considering going into an unvisited state (in case where the vertex hasn't been discovered but is not allowed to be visited because the corresponing move is tabue'd. To handle this, we can store extra info about each move → $(a_i, b_i, f_i, f_i + 1)$ where a_i and b_i are slots being swapped and f_i is obj function before swapping and $f_(i + 1)$ the is obj function value after swapping.
- Alternatively, we can impose a restriction that we cannot change the value of a decision variable for some time.

Summarizing with an alternate view

TRIVIAL ALGORITHMS

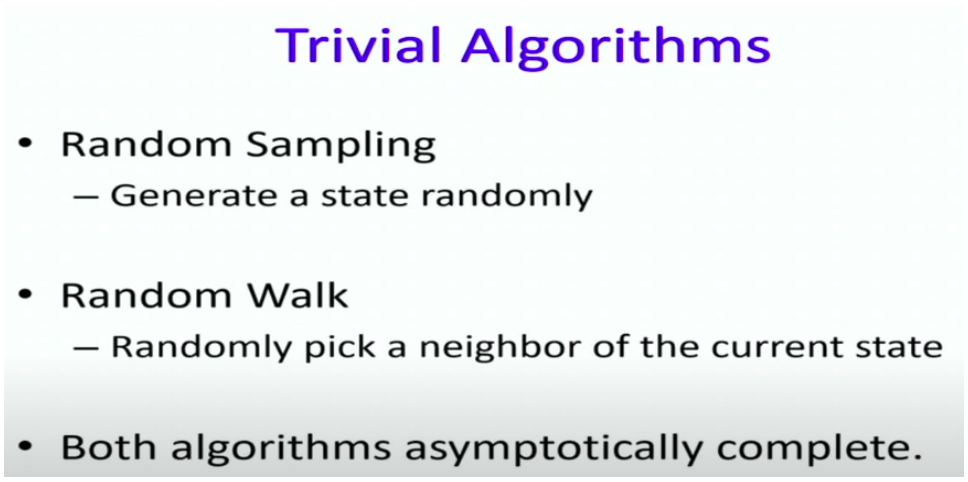
Proposal 1: Random sampling: (Asymptotically complete)

Keep generating solutions randomly unless a desired soln is found.

Eg in the n-queens problem → A random soln can be generated by randomly choosing a row for 1st queen, then randomly choosing a row for 2nd queen and so on. The thing here is that we keep generating solutions randomly without keeping track of the previous states and hope to find an optimal solution.

Proposal 2: Random walk (Asymptotically complete)

EgL in n queens problem, one state has 56 other neighbours $(8 * 7)$ and we randomly choose one of the 56 neighbours and keep doing this until an optimal solution is found.



BOTH ABOVE PROPOSALS seem to be ineffective as they don't seem to make informed decisions and randomisation does not make complete sense given neighborhood is not particularly large.

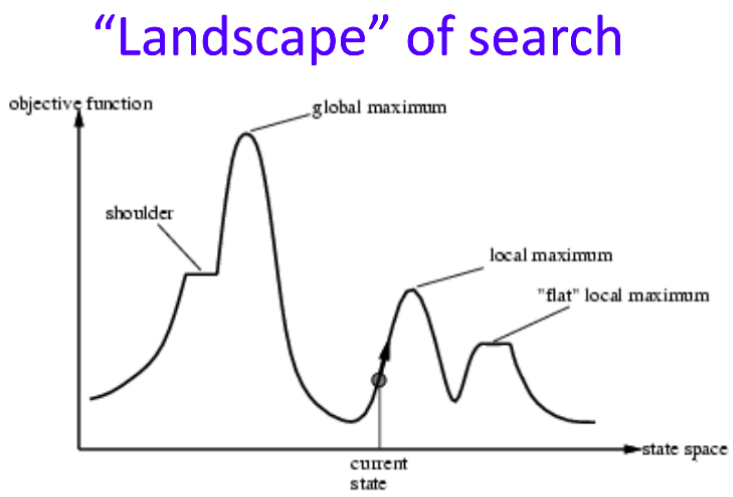
Proposal 3: HILL CLIMBING (GREEDY LOCAL SEARCH)

(NOT Asymptotically complete as we might get stuck in a local optima)

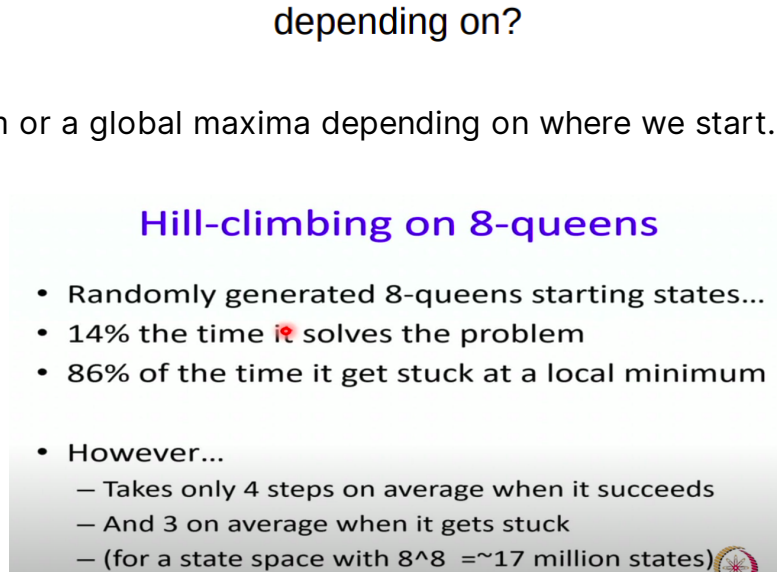
Out of the 56 neighbours of a particular state in the n-queens problem, I greedily choose the one which has the best value of the optimization function. Keep doing this until all the neighbours of a current state are less desirable than current state (as judged by value of optimization function).THis method can be described as a **person climbing mount everest in a foggy path while having amnesia.**

- The person hopes to reach the top (as does the algorithm)
- **Foggy** → A person (state) can only see its immediate nearby climbers (neighbours)
- **Amnesia →** The algorithm has no memory how it reached the current state

But we might get stuck at a local maxima

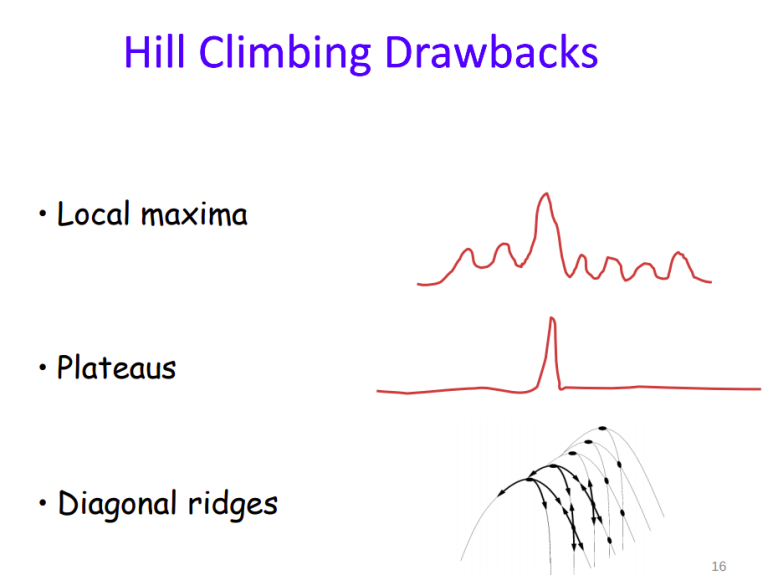


We may reach a local maximum or a global maxima depending on where we start.



HILL CLIMBING DRAWBACKS

- **Local maxima**
- **Plateaus:: Neighbours have the same cost even though current state is not even a local maxima**
- **Diagonal ridges:**



NOW, if I am stuck on a local maxima and I want to go the global maxima, then :

We have to move to states lower than the current local maxima in order to have any chance of making it to the global maxima.

Escaping plateaus: Allow sideways moves as well

We can escape from a plateau like situation by allowing sideways moves as well. So, instead of just climbing uphill, we also allow sideways moves (ie $f(current_state) = f(neighbour)$ can also make us make a move to the neighbour if no better neighbour exists).**BUT EVEN HERE, we might be moving in a directed cycle of states that have the same value for the optimization function and hence, it is important we limit the number of sideways moves we make.**

Escaping Shoulders: Sideways Move

- If no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape
  - Need to place a limit on the possible number of sideways moves to avoid infinite loops
- For 8-queens
  - Now allow sideways moves with a limit of 100
  - Raises percentage of problem instances solved from 14 to 94%
  - However….
    - 21 steps for every successful solution
    - 64 for each failure

UP UNTIL NOW, we had total amnesia and remembered nothing about the states we have already explored in the past.

- **We can fix this problem and keep a TABU list.**

Tabu Search

- prevent returning quickly to the same state
- Keep fixed length queue (“tabu list”)
- add most recent state to queue; drop oldest
- Never make the step that is currently tabu’ed

- Properties:
  - As the size of the tabu list grows, hill-climbing will asymptotically become “non-redundant” (won’t look at the same state twice)
  - In practice, a reasonable sized tabu list (say 100 or so) improves the performance of hill climbing in many problems

---

ENFORCED HILL CLIMBING: MIDDLE GROUND OF LOCAL SEARCH AND SYSTEMATIC SEARCH.

Alternatively, after reaching a local optima, we can start a BFS with the hopes of striking a state which is better than the current local maxima (root) and with the potential to keep moving up. Thus, the moment we are able to escape from the local maxima by means of BFS by reaching a node better than the root, we start HILL CLIMBING AGAIN.

IF THE local maxima we had reached turns out to be the global maxima (which we are not aware of) our algorithm keeps running until we run out of time/memory.

Escaping Shoulders/local Optima

Enforced Hill Climbing

- Perform breadth first search from a local optima
  - to find the next state with better h function

- Typically,
  - prolonged periods of exhaustive search
  - bridged by relatively quick periods of hill-climbing

- Middle ground b/w local and systematic search

UP UNTIL NOW, two main types of algos were summarized:

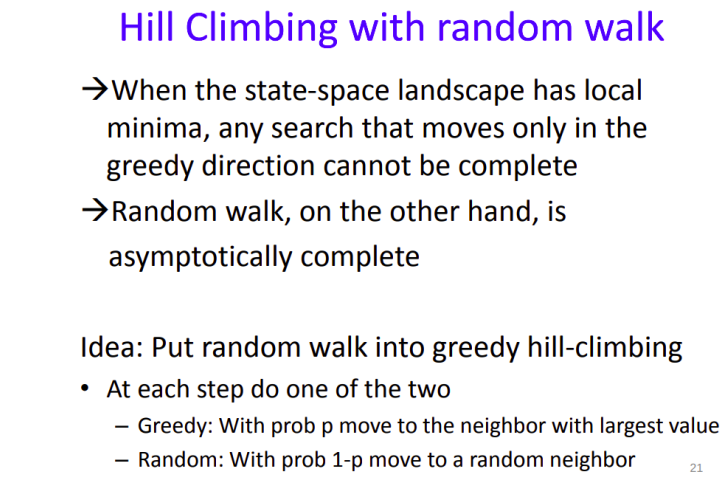
- Asymptotically complete(random search, random walk)
- Not Asymptotically complete (hill climbing)

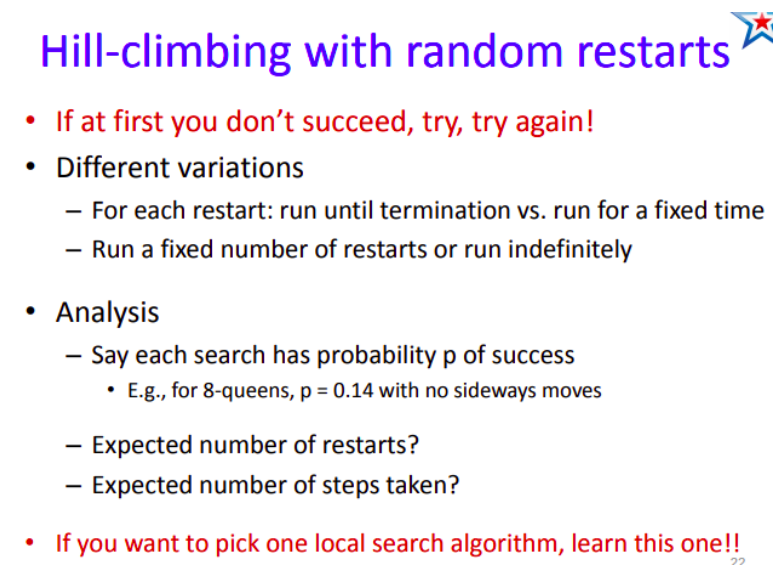
WE CAN TRY TO FORM A HYBRID OF THE TWO.

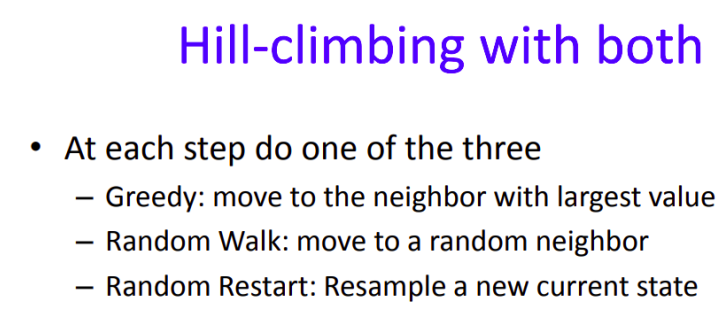
STOCHASTIC VARIATIONS OF HILL CLIMBING

- Whenever stuck on a local maxima /plateau, randomly sample a new neighbour. **(GREEDY + RANDOM WALK)**
- Whenever stuck on a local maxima/plateau, randomly sample a new state (not necessarily a neighbour of the current state) **[GREEDY + RANDOM Sampling]**

OR







The most famous algo which achieves (greedy + random) is simulated annealing algorithm (asymptotically optimal → in the limit of infinite time, we are sure to reach a global optimum)

Just pick a random neighbour N first and then decide whether to move to this neighbour or not. If a random neighbour N is more desirable than current state C in terms of obj function, go to N surely, else go to N with some probability (depending on how bad it is and in which stage of the search we are).

> A thorough discussion of simulated annealing technique with application in solving the TSP problem can be found in the TSP report.

Intuition for simulated annealing:

Assume we are starting from the top of a hill and the goal is to reach to the bottom at the river. Now, if we get stuck at the local minimum between two hills, simulated annealing is analogous to an earthquake which helps us to escape the local minima.

Guided LS and Fast LS Meta-Heuristics

Reference: https://www.bracil.net/CSP/papers/VouTsa-Gls-MetaHeuristic2003.pdf
Guided local search is a penalty-based metaheuristic algorithms that wraps around an existing local search algorithm. It is a way to come out of local optimum to continue exploring.

Fast local search on the other hand is a way to reduce the neighborhood size explored in each individual move while maintaining the same overall neighborhood size/connectivity.

Guided Local Search

Outline

For GLS, one defines a set of features present in the candidate solutions. When LS gets trapped in a local optima, certain features are selected and penalized. The objective function to be evaluated during local moves is augmented with these feature penalties. These are described in more detail below. Care must be taken that too much penalties are not built up during the execution, because this could lead the algorithm away from good solutions. This can lead to the rate of improvement dropping drastically over time in GLS.

GLS is closely related to taboo search as described earlier. It is a softer taboo that guides away from local minima, without completely 'preventing' them from recurring. In-fact, the idea of penalizing transitions instead of states (taboo lists) can be used when features for states become too large. Moreover, ideas related to aspiration (taking taboo moves if the improvement is significant) can also be applied in a more advanced implementation.

Explanation using TSP

Let's take the travelling salesman problem as a running example to explain the procedure.

A possible 'feature' here could be whether the edge 'A→B' exists in the current candidate solution. Each feature has an associated 'cost' and 'penalty'. The cost here is often defined to be the same as the objective function. Eg: in TSP, the 'distance'/'weight' on the edge. The penalties for each feature are initialized 0 and increased eventually.

General Formulation

GLS defines a function h that will be used for evaluating local moves: $h(s) = g(s) + \lambda * (\sum p_i * I_i(s))$
Here g is the original cost function, λ is a coefficient for penalty weighting, p_i is the penalty associated with the i^{th} feature and $I_i(s)$ is 0 or 1 depending on whether the i^{th} feature is present in the current state s.

How are penalties calculated? GLS aims to penalize *unfavorable features* in the local optima that *matter most*. Everytime a local optima is encountered, the feature with the highest $util$ value sees an increment of 1 in it's penalty.
$util_i = I_i(s) * \frac{c_i}{1+p_i}$
where c_i is the cost of the i^{th} feature (how unfavorable it is, eg: edge-length in TSP).

The intuition is that the higher the cost of the feature, the more is the utility of penalizing it, and the more a feature has been already penalized, the less the utility of penalizing it again.

So while GLS moves towards local optima (as they have lower h values), it comes out of them quickly due to the penalties that get applied.

The performance of GLS is not too sensitive to λ on some problems, but on others it may be so. A recommended starting point before tuning is:
$\lambda = \alpha * g(s_)/NumFeaturesIn(s_)$ where α is now a constant that can be tuned based on the data-instance and s_* is the local minima.

Pseudocode

```
procedure GuidedLocalSeach(p, g, λ, [I₁, ...,Iₘ], [c₁,...,cₘ],
M)
begin
     k ← 0;
```

```
    s0 ← ConstructionMethod(p);
    /* set all penalties to 0 */
    for i ←1 until M do
        pi ← 0;
    /* define the augmented objective function */
    h ← g + λ * ∑pi*Ii ;
    while StoppingCriterion do
    begin
        sk+1 ← ImprovementMethod(sk, h);
        /* compute the utility of features */
        for i ←1 until M do
            utili ← Ii(sk+1) * ci / (1+pi);
        /* penalize features with maximum utility */
        for each i such that utili is maximum do
            pi ← pi + 1;
        k ← k+1;
    end
    s* ← best solution found with respect to objective
function g;
    return s*;
end
```

Fast Local Search

Guided local search works best with fast methods to find local minima so that more iterations of penalties can be applied with small rate of scaling up. While fast local search is an independent technique in and of itself, it's thus often coupled with guided local search.

Outline

The neighbourhood is broken down into small sub-neighbourhoods that seem natural for the problem. An activation bit is assigned to each sub-neighbourhood. Initially, all are active. Neighborhoods are examined in a fixed order. If on examining a particular sub-neighborhood no improving moves are found, it is turned into inactive. It is made active again when an *adjacent* neighborhood is penalized because then this neighborhood's situation is likely to have changed. Once all neighborhoods become inactive, we can say the iteration of local search has completed. From here, we can choose to re-start or try some different initialization.

Explanation Using TSP

Example, in TSP with 2-opt, we initially have an $O(N^2)$ neighborhood of all possible sub-segment reversals. A sub-neighborhood can be a particular node which has to be considered for swaps. Whenever an edge incident on the node is penalized (if using guided local search too), the node is again considered as an end-point for sub-segment reversal.

Pseudocode

```
procedure FastLocalSeach(s, h, [bit1, …,bitL], L)
begin
    while ∃bit, bit = 1 do
    /* i.e. while active sub-neighborhood exists */
        for i ←1 until L do
        begin
            if biti = 1 then
            /* search sub-neighborhood i */
            begin
                Moves ← MovesForSubneighborhood(i);
                for each move m in Moves do
                begin
                    s′ ← m(s);
                    /* s′ is the result of move m */
                    if h(s′) < h(s) then
                    /* minimization case is assumed here */
                    begin
                        /* spread activation */
                        ActivateSet ←
                        SubneighborhoodsForMove(m);
                        for each sub-neighborhood j
                        in ActivateSet do
                            bitj ← 1;
                        s ← s′;
                        goto ImprovingMoveFound
                    end
                end
                biti ← 0; /* no improving move found */
            end
ImprovingMoveFound:
            continue;
        end;
    return s;
end
```

Guided Fast Local Search

So let's see how these two techniques look when combined:

Combined Pseudocode

```
procedure GuidedFastLocalSearch(p, g, λ, [I1, ...,IM],
[c1,...,cM], M, L)
begin
    k ← 0;
    s0 ← ConstructionMethod(p);
    /* set all penalties to 0 */
    for i ←1 until M do
        pi ← 0;
    /* set all sub-neighborhoods to the active state */
    for i ←1 until L do
        bit_i ← 1;
/* define the augmented objective function */
h ← g + λ * ∑pi*Ii;
while StoppingCriterion do
begin
    sk+1 ← FastLocalSearch(sk, h, [bit1, …,bitL], L);
    /* compute the utility of features */
    for i ←1 until M do
        utili ← Ii(sk+1) * ci / (1+pi);
    /* penalize features with maximum utility */
    for each i such that utili is maximum do
    begin
        pi ← pi + 1;
        /* activate sub-neighborhoods related
        to penalized feature i */
        ActivateSet ← SubneighborhoodsForFeature(i);
        for each sub-neighborhood j in ActivateSet do
            bitj ← 1;
    end
    k ← k+1;
end
s* ← best solution found with respect to objective
function g;
return s*;
end
```