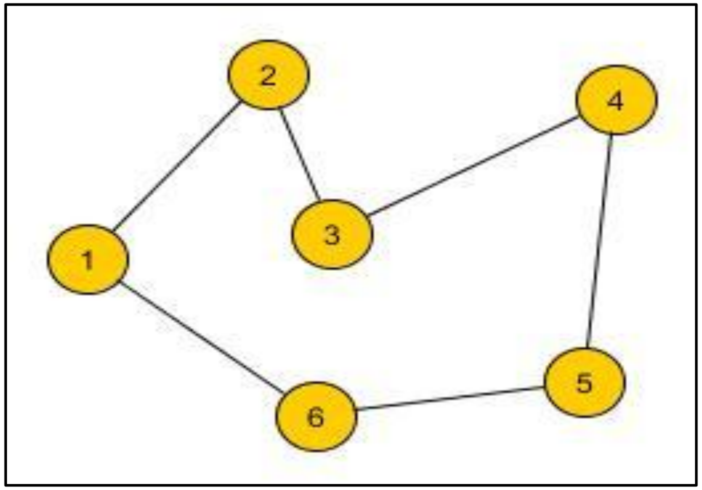
CSE 402
Offline 3
(A1 & B1)

Local Search

- Technique used for difficult optimization problem or constraint satisfaction problem.
- For some problem, we don’t need the path to the solution

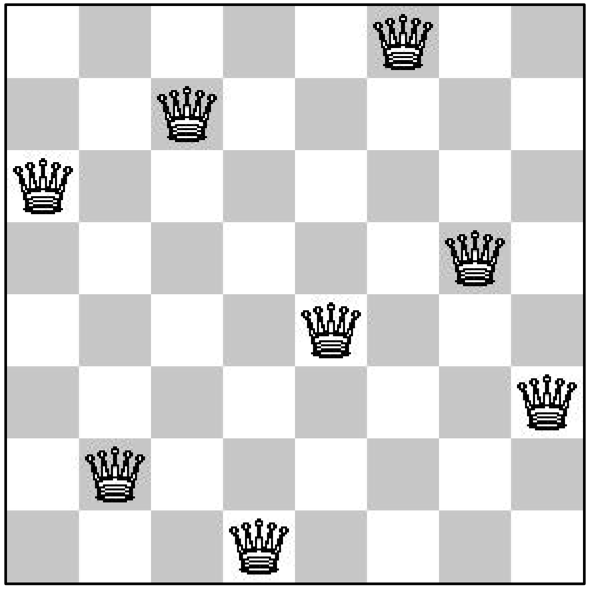
Traveling Salesperson Problem(TSP)

- Given *n* points/cities and distance between any two pair of cities.
- Can you find a tour with minimum distance visiting each city?



N queens Problem

- How can *n* queens be placed on an *N*x*N* chessboard so that no two of them attack each other?



Key Idea

- A local search algorithm usually looks like following
  1. Pick an initial state (Randomly or using some heuristics)
  2. Make local modification to improve current state
  3. Repeat step 2 until goal state found (or out of time)

Some Issues

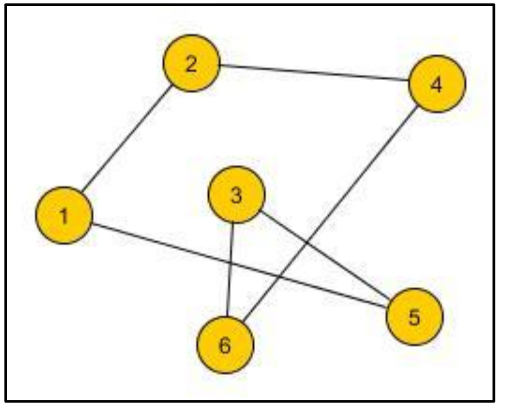
- What is a state?
- How you define local?
- How to measure improvement?

What is a state?

- State can be partial or complete solution
- Usually a complete solution

Solution in TSP

- Solution in TSP may be a tour.



- We can represent it using a vector of integers.

<1,2,4,6,3,5>

Solution in *n*-queens

- Solution in TSP may be an assignment of queens in the chessboard.

		Q	
	Q		
			Q
Q			

- We can represent it using a 2d array or 1d array

0 0 1 0

0 1 0 0

0 0 0 1 <3,2,4,1>

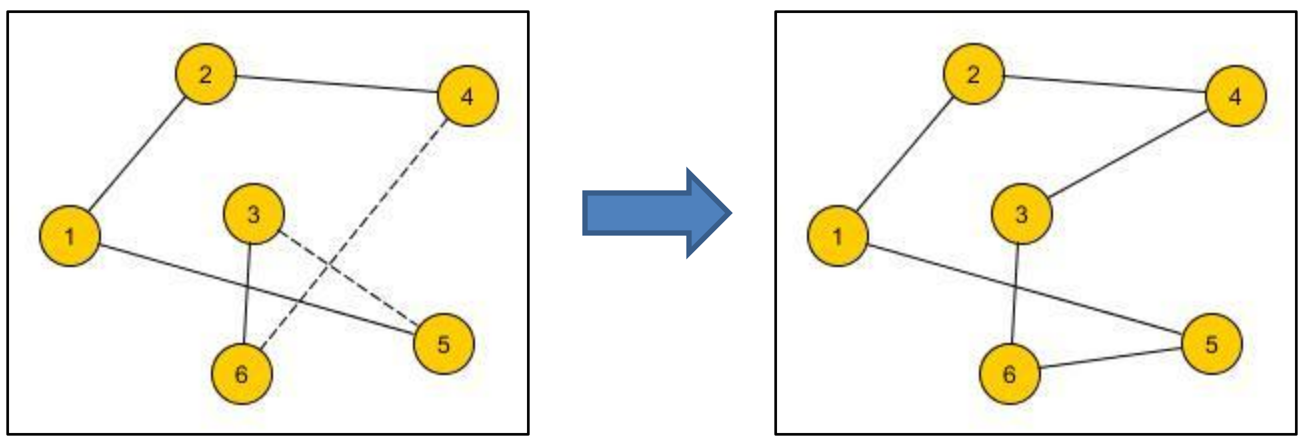
1 0 0 0

How you define local?

- Small change in current state/solution.
- Usually we define one or more neighborhood function(s).

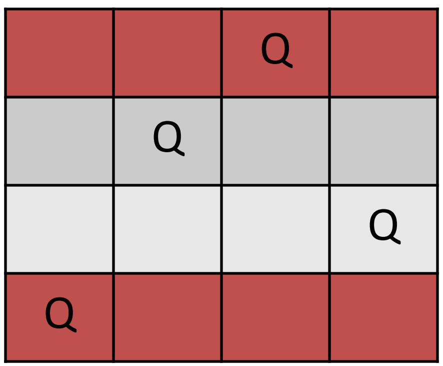
Neighborhood in TSP

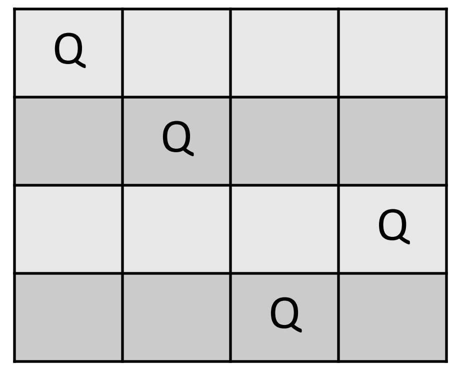
- 2-opt operator:



Neighborhood in *n*-queens

- Swap two rows:





How to measure improvement?

- For optimization problem we have a objective function to optimize.
- For CSP, we can use a heuristic function.

How to measure improvement?

- In TSP:
  - Cost of current tour. (Improvement means reduction of cost)
- In *n*-queens:
  - # of conflicts. (Improvement means less number of conflicts)

Hill Climbing

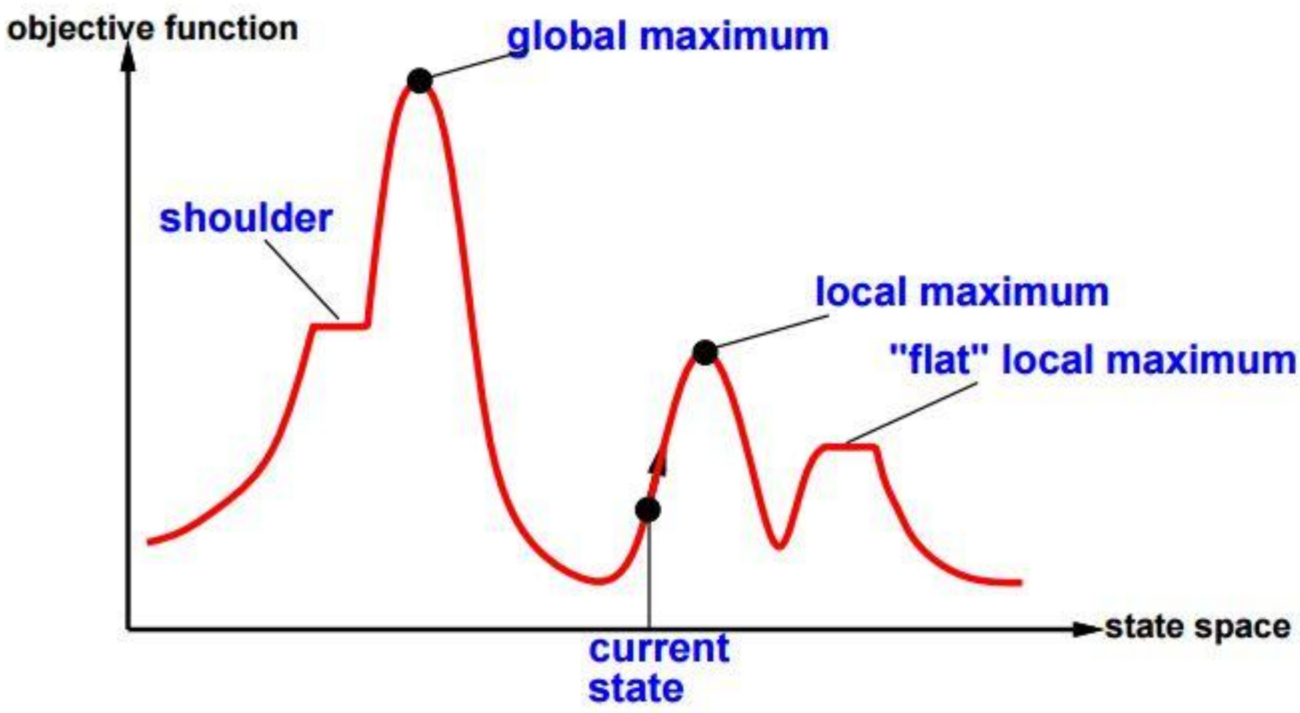
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
       neighbor ← a highest-valued successor of current
       if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
       current ← neighbor
   end
```

- At each step, move to a neighbor of higher value in hopes of getting to a solution having the highest possible value
- Can easily modified for minimization problem

Hill Climbing

- Can stuck into local optima



Variations

- Steepest ascent hill climbing:
  - Take best neighbor
- Stochastic hill climbing:
  - Select random better neighbor
- First choice hill climbing:
  - Take first better randomly generated neighbor
- Random restart hill climbing:
  - Restart again from random state if you stuck in local optima

Simulated Annealing

- Basic Idea:
  - Like hill climbing identify the quality of local improvements
  - Assume that change in objective function is δ
  - If δ is positive, move to that state
  - Otherwise move to that state with a probability proportional to δ and *T(?)*
  - Over time make it less likely to accept bad moves.

Simulated Annealing

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $i \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[i]
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* ← *next*
    else with probability $e^{\Delta E/T}$, set *current* ← *next*

Scheduling function

- T should be large at beginning and gradually decrease
- If T is lowered slow enough, then we can reach global optima
- What do you mean by slow enough?
- In literature different function can be found:
  - Linear Cooling: $schedule(T) = T_0 - \mu t$
  - Exponential Cooling: $schedule(T) = T_0\, \alpha^t$, $0 < t < 1$
  - Logarithmic Cooling: $schedule(T) = c / log(1 + t)$

Offline-3

Offline

- You have to implement local search algorithms for *permutation flow-shop scheduling* problem.

- **Algorithms**:
  - First Choice Hill Climbing
  - Simulated Annealing

Permutation Flow-shop Scheduling

- A set $J=\{1,2,3,\ldots,n\}$ of n independent jobs
- A set $M=\{1,2,3,\ldots,m\}$ of m machines
- Each job has exactly m tasks, where i^{th} task is performed by the i^{th} machine
- Processing time of i^{th} job in j^{th} machine is $P_{i,j}$
- Every job goes through a predefined route

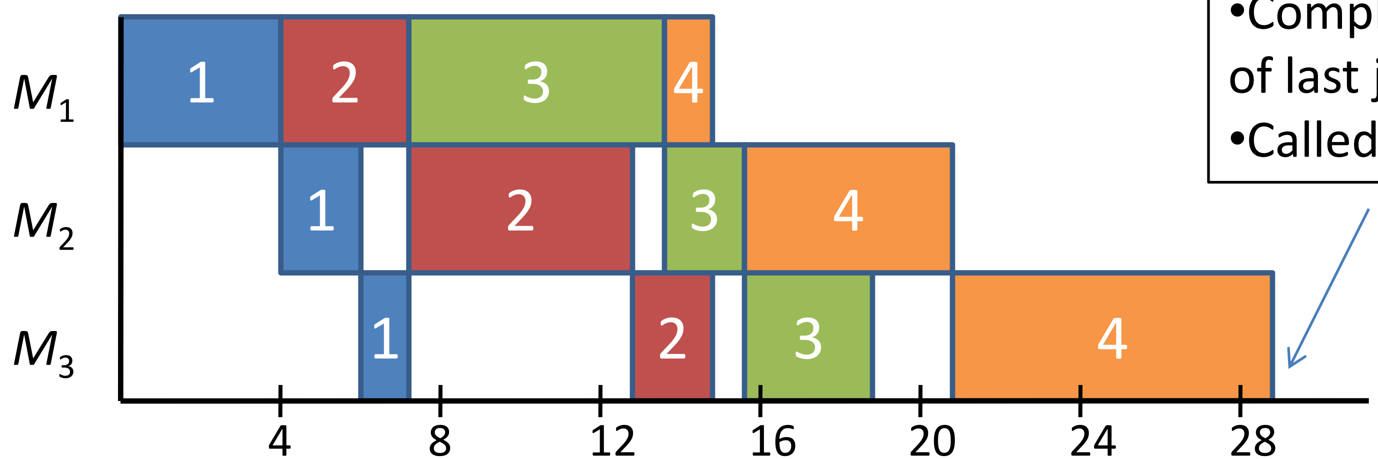
  Machine 1->2->3->4->……->m

- All jobs arrive at time 0

Permutation Flow-shop Scheduling

j	$P_{1,j}$	$P_{2,j}$	$P_{3,j}$
1	4	2	1
2	3	6	2
3	7	2	3
4	1	5	8

For a permutation schedule 1, 2, 3, 4



- Completion time of last job=29
- Called **makespan**

Permutation Flow-shop Scheduling

- There are different Objective functions in literature
- We will use makespan
- Formally,
  - Let C_i be the completion time of i^{th} job
  - Makespan: $\max(C_i \mid i \in \{1,2,3,\ldots,n\})$
- Find a permutation that minimizes makespan

Representation

- A solution is a permutation.

$$\pi = < \pi_1, \pi_2, \pi_3, \dots , \pi_n >$$

- Can be represented by a vector of length *n*.

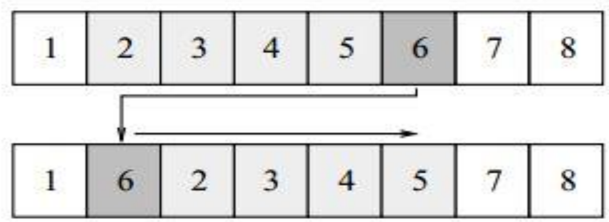
Evaluation Function

- Each solution is evaluated by its makespan
- Let $C_{\pi(i),j}$ be the completion time of j^{th} task of $\pi(i)$ job. So makespan= $C_{\pi(n),m}$

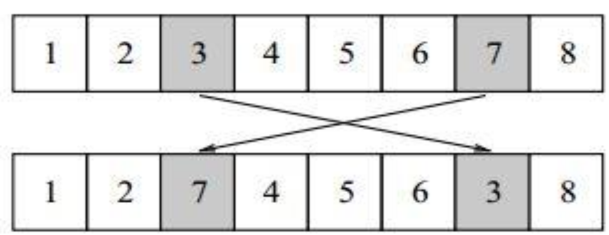
$$C_{\pi(i),j} = \begin{cases} P_{\pi(i),j} & \text{if } \pi(i) = j = 1 \\ C_{\pi(i-1),j} + P_{i,j} & \text{if } j = 1 \text{ and } \pi(i) > 1 \\ C_{\pi(i),j-1} + P_{i,j} & \text{if } j > 1 \text{ and } \pi(i) = 1 \\ \max(C_{\pi(i-1),j}, C_{\pi(i),j-1}) + P_{i,j} & \text{otheriwse} \end{cases}$$

Neighborhood Function

- Insertion Operator:
  - An element of a random position is removed and put into another position



- Exchange Operator:
  - Two randomly selected elements are swapped.



i) First Choice Hill Climbing

Algorithm:

```
current ← randomly generated state
max_step← maximum number of iteration
count ← 1
 for( i ← 0 ; i < max_step ; i++, count++) {
     for(j ← 0; j< t ; j++){
        if( a random number between [0,1] < 0.5 )
                      neighbor← a successor of current state by insertion operator
        else
                      neighbor← a successor of current state by exchange operator
        if( makespan(neighbor) < makespan(current) )
                      current← neighbor; break;
     }
     if( j= t) return current
 }
return current
```

ii)Simulated Annealing

- You have to implement Simulated Annealing algorithm shown previously.
- You will use linear cooling scheme as scheduling function.
- Neighborhood and Evaluation function is same as shown in Hill climbing approach.

Main function

- You will have to run hill climbing and simulated annealing for same initial state.

```
for(i=1 to 10)
        Solution x = GenerateRandomState()
        HillClimb(x)
        SimulatedAnnealing(x)
```

Output

- You have to run your code for given data sets.
- For each data set run your program ten times.
- Generate following table for and submit it along with source code.
- Also discuss the result.

Data Set	t	Average No of Iteration (HC)	Average Makespan (HC)	Minimum makespan (HC)	Average No of Iteration (SA)	Average Makespan (SA)	Average Makespan (SA)

Submission Deadline

- Deadline is 15 November 2016, 2:00 AM
- And Please Do not Copy

- During evaluation you have to show both makespan value and corresponding schedule. For a data set we will run hill climbing and simulated annealing with same initial state