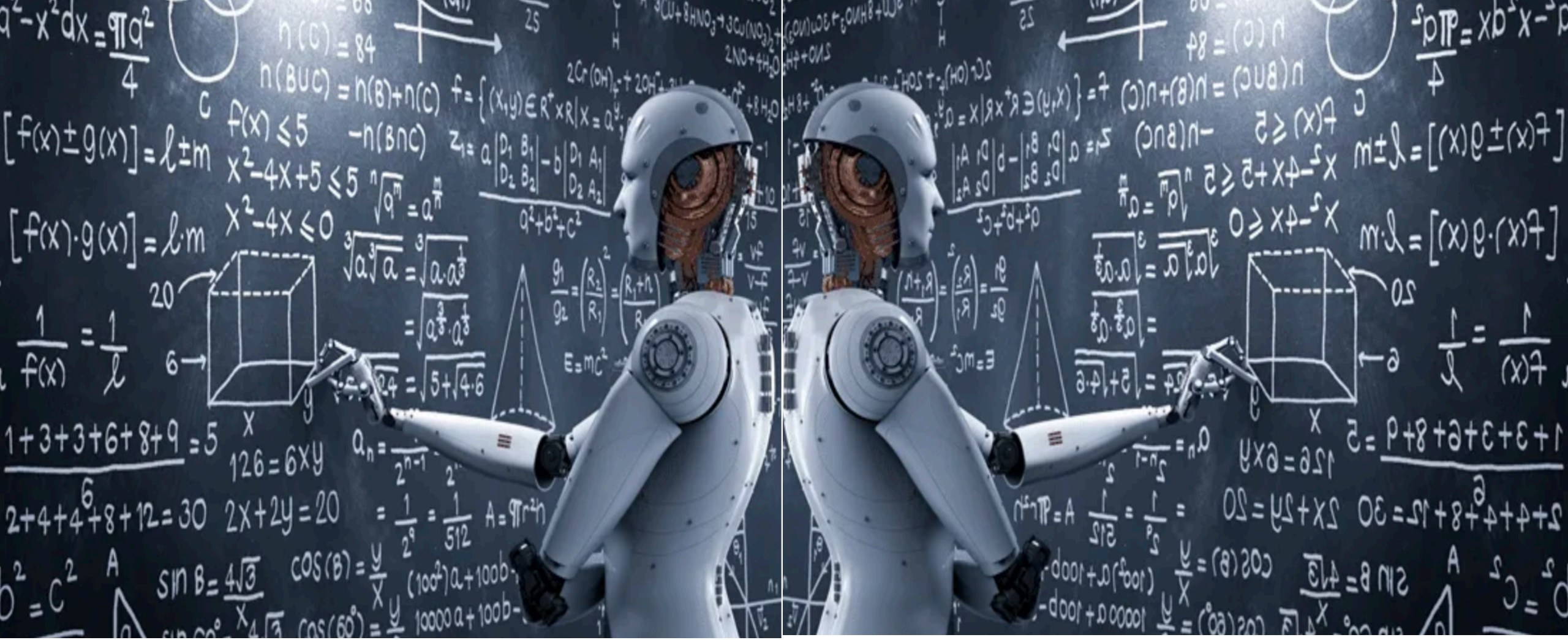
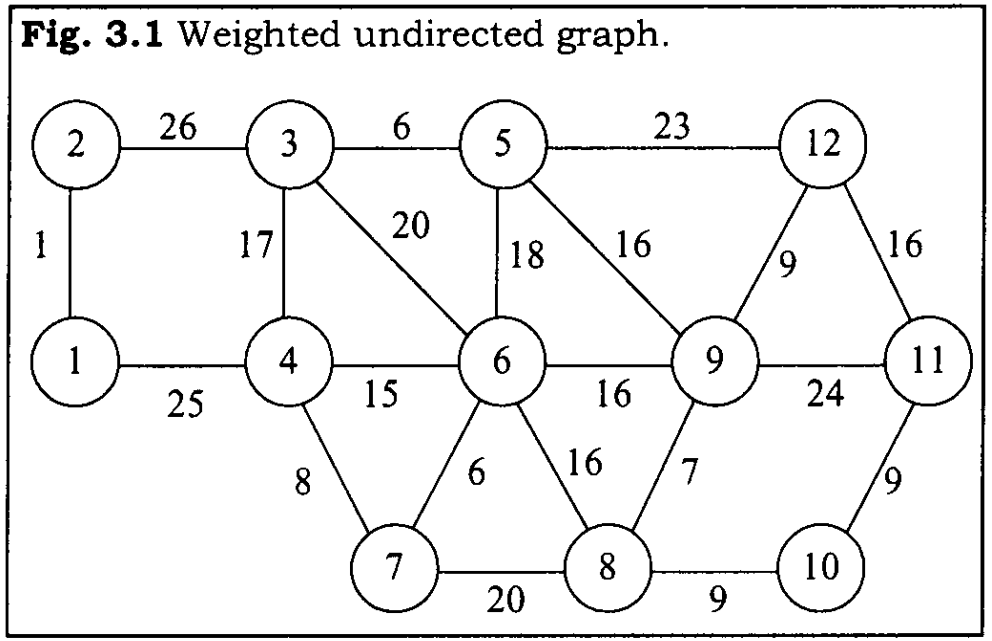


TABU SEARCH

Examples

MINIMUM K-TREE

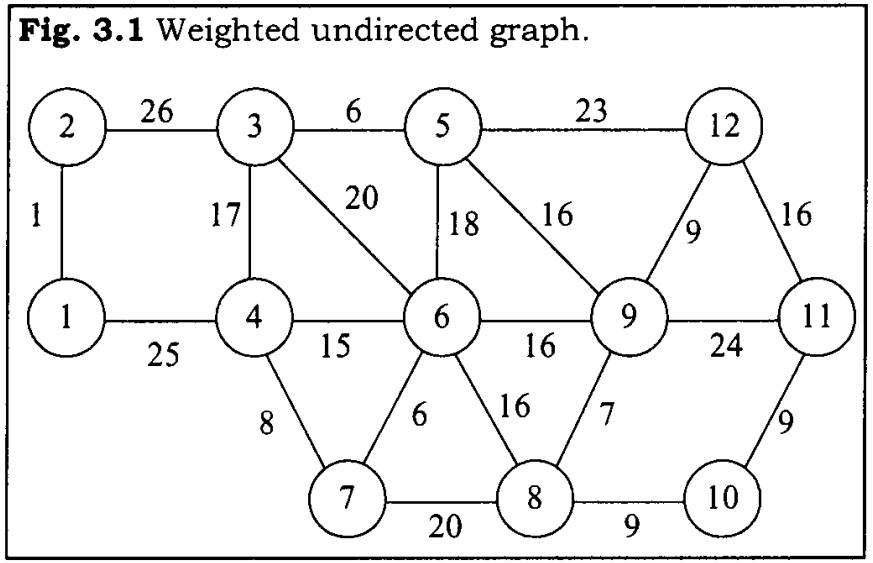
Problem definition: Given a graph, *find the tree consisting of k-edges of the graph such that the sum of the edge weights is minimum*



EXMPLE 1: GREEDY CONSTRUCTION OF INITIAL K-TREE (K=4)

Table 3.1 Greedy construction.

Step	Candidates	Selection	Total Weight
1	(1,2)	(1,2)	1
2	(1,4), (2,3)	(1,4)	26
3	(2,3), (3,4), (4,6), (4,7)	(4,7)	34
4	(2,3), (3,4), (4,6), (6,7), (7,8)	(6,7)	40

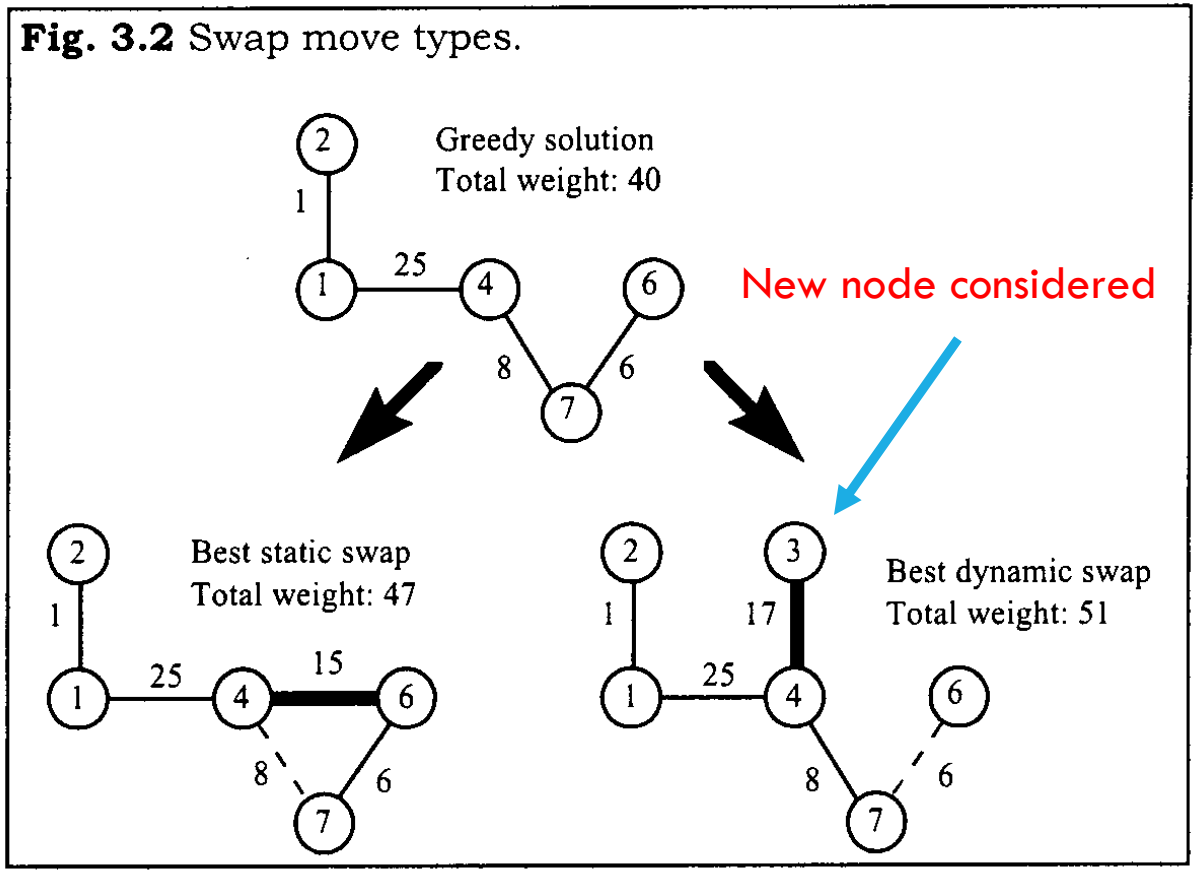


- Start with the edge having the minimum weight
- Choose remaining *k*-1 edges successively to minimize the increase in total weight in each step

EXAMPLE 1: NEIGHBORHOOD STRUCTURE

- Basic move is edge swapping
- Consider all feasible swaps that result in a tree and that are not tabu (in general, infeasible moves may be allowed occasionally)
- Choose the move yielding the minimum total weight
- A static swap keeps current nodes, a dynamic one allows change of nodes

EXAMPLE 1: NEIGHBORHOOD STRUCTURE (CONT.)

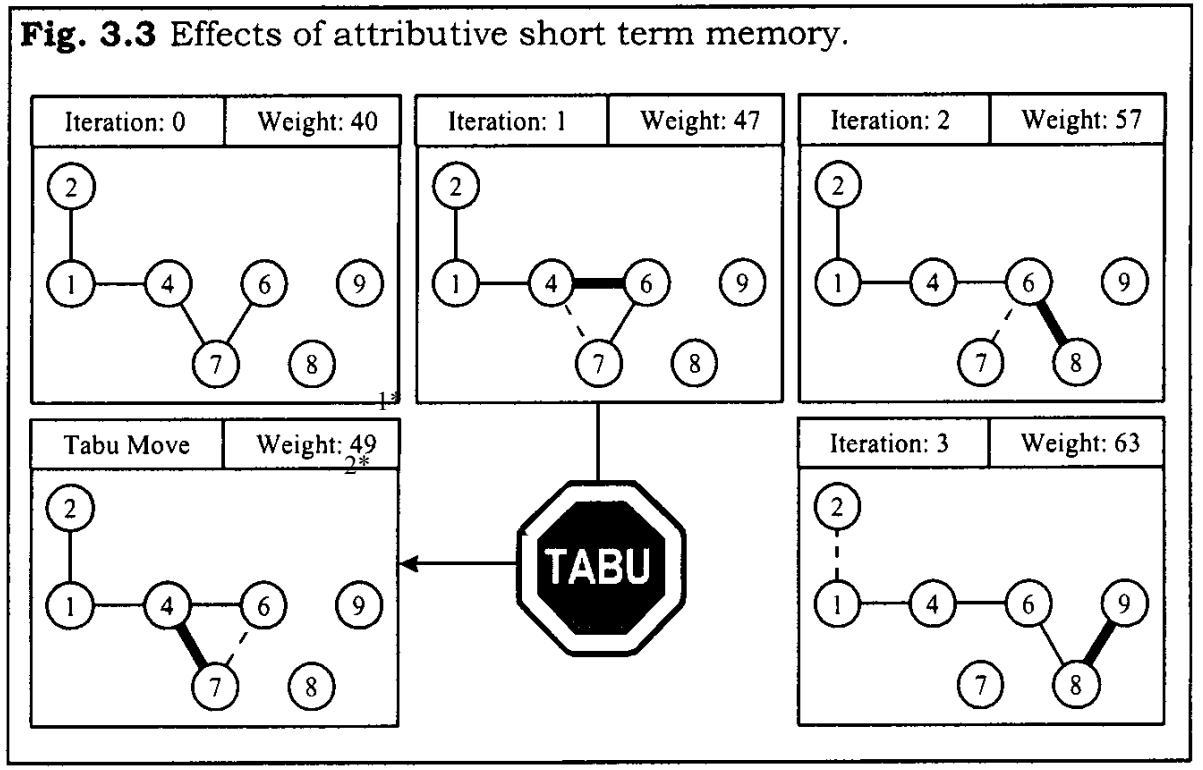


Static swap is selected

EXAMPLE 1: TABU CLASSIFICATION

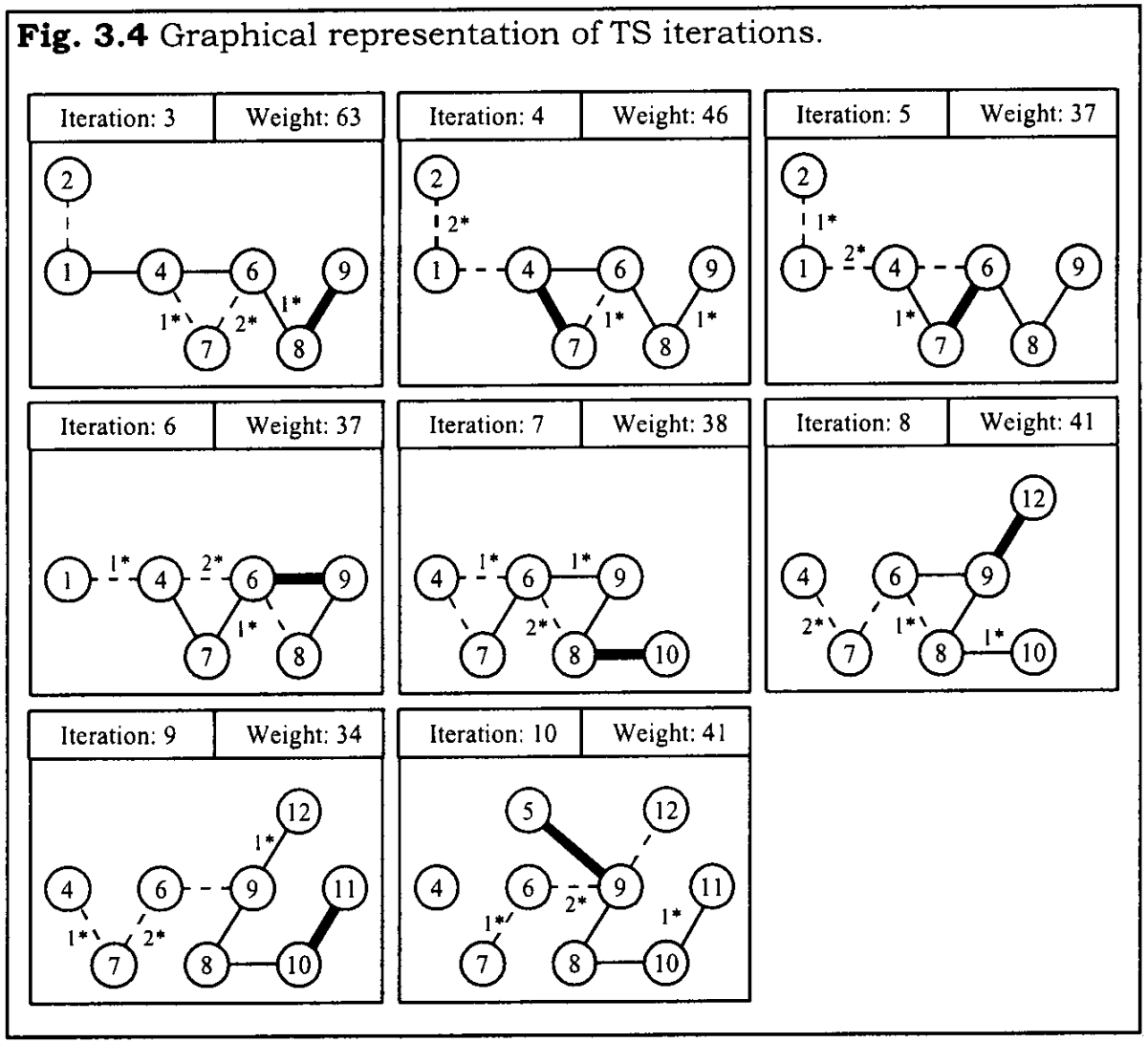
- Tabu attributes are edges swapped
- Out-edge is $\{4,7\}$, in-edge is $\{4,6\}$ in Figure 3.2
- Tabu tenure for in-edges (bounded by k) is set as 1
- Tabu tenure for out-edges is chosen as 2 (since there are more outside edges)
- A swap move is classified as tabu if either the out-edge or the in-edge involved is tabu-active (an alternative rule could have been "if both are tabu-active")

EXAMPLE 1: TABU CLASSIFICATION MAY PREVENT VISITING UNEXAMINED SOLUTIONS



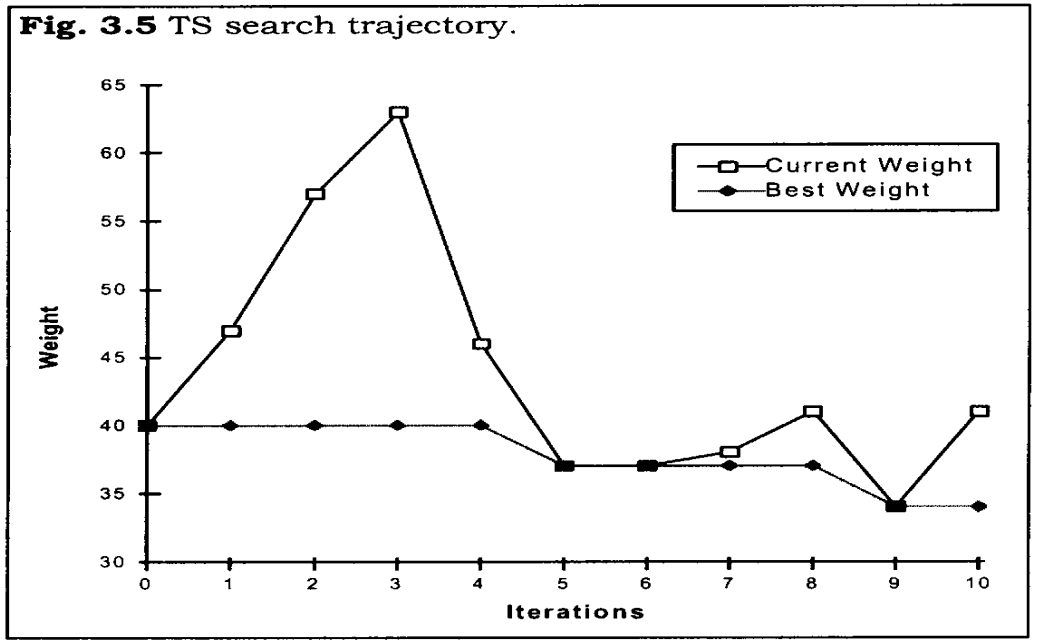
In iteration 2, in-edge {4,7} and out-edge {6,7} with objective function value 49 cannot be visited since {4,7} is tabu-active, and we end up with an inferior solution

ADDITIONAL ITERATIONS



34 is the optimal solution

EXAMPLE 1: TS SEARCH TRAJECTORY



Note the local optima in iterations 5 and 6

DIVERSIFICATION

- We may need to diversify or restart TS when, for example, no admissible improving moves exist or rate of finding new best solutions drops
- Instead of choosing the restarting point randomly, TS employs diversification strategies based on:
  - Frequency based memory: records frequently used attributes or visited solutions
  - Critical event memory: an aggregate summary of critical events (local optima or elite solutons) during the search

DIVERSIFICATION: FREQUENCY BASED MEMORY

Iteration 26
Current solution

1	3	6	2	7	5	4

Insulation Value=12

Tabu structure
(Recency)

	1	2	3	4	5	6	7
1	■			3			
2		■					
3	3		■			2	
4	1	5		■			1
5		4		4	■		
6			1		2	■	
7	2			3			■

(Frequency)

Top 5 candidates

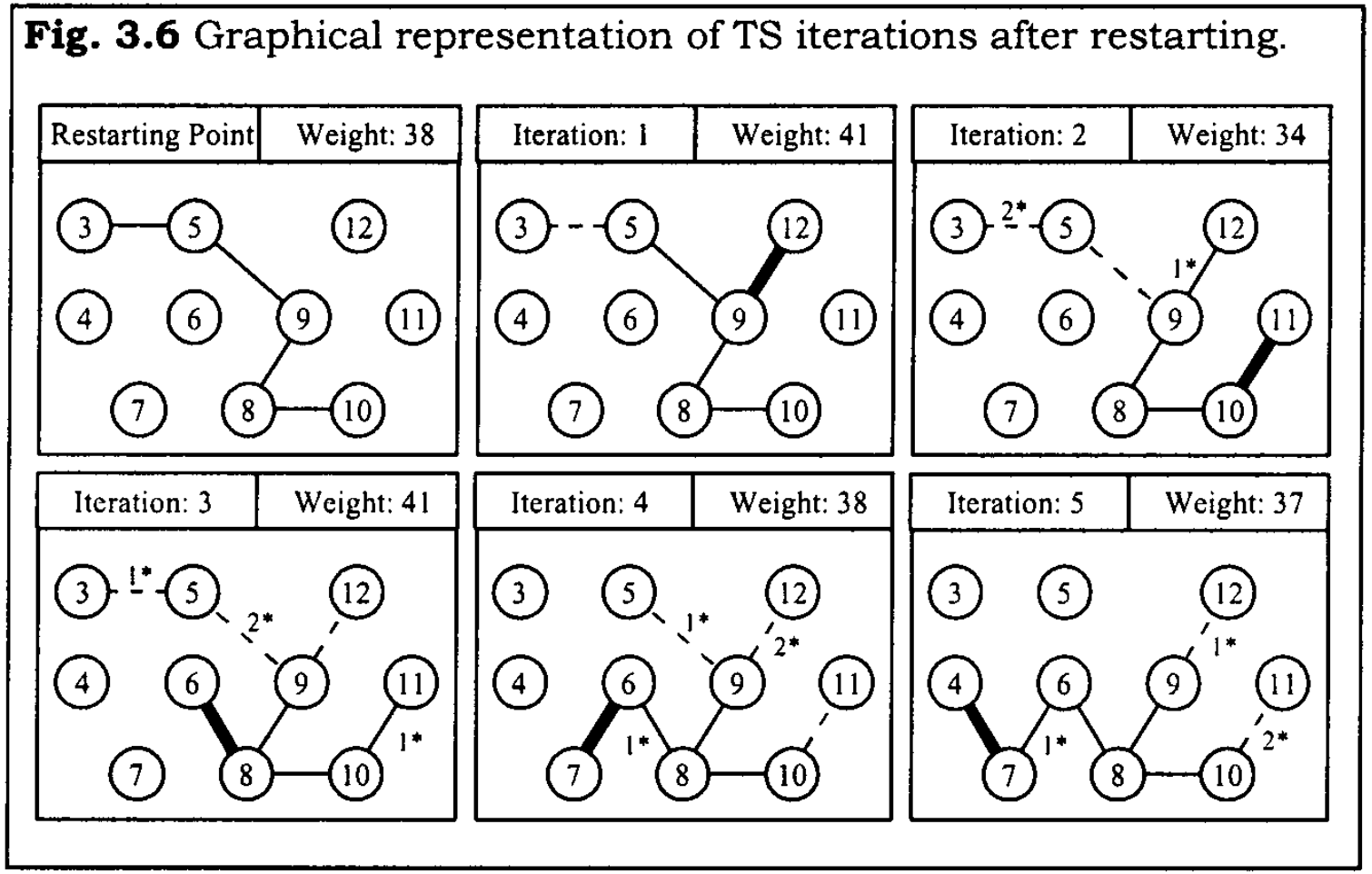
Swap	Value	Penalized Value	
1,4	3	3	T
2,4	-1	-6	
3,7	-3	-3	*
1,6	-5	-5	
6,5	-4	-6	

- ✓ Diversify when no admissible improving moves exist
- ✓ Penalize non−improving moves by assigning larger penalty to more frequent swaps, choose $(3,7)$ using penalized value.

DIVERSIFICATION

- Summary of the starting solution and local optima (elite solutions) found in previous start(s)
- Assuming we choose to restart after iteration 7, these are the starting solution (40), and solutions found in iterations 5 (37) and 6 (37)
- In finding the restarting solution, penalize the use of edges in these solutions for diversification
- May use frequency based memory (frequency of appearance of these edges) for weighing the edges

EXAMPLE 2: RESTART ITERATIONS



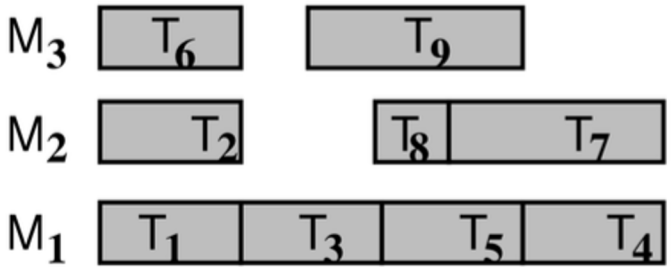
TASK SCHEDULING

Scheduling

concerns optimal allocation or assignment of resources, to a set of tasks or activities over time

- limited amount of resources,
- gain maximization given constraints

- Machines $M_i, i = 1, \ldots, m$
- Jobs $J_j, j = 1, \ldots, n$
- (i, j) **an operation** or processing of jobs j on machine i
  - a job can be composed from several operations,
  - example: job 4 has three operations with non-zero processing time (2,4),(3,4),(6,4), i.e. it is performed on machines 2,3,6



- Static parameters of job
  - **processing time** p_{ij}, p_j:
    processing time of job j on machine i
  - **release date of** j r_j:
    earliest starting time of jobs j
  - **due date** d_j:
    committed completion time of job j (preference)
  - vs. **deadline**:
    time, when job j must be finished at latest (requirement)
  - **weight** w_j:
    importance of job j relatively to other jobs in the system
- Dynamic parameters of job
  - **start time** S_{ij}, S_j:
    time when job j is started on machine i
  - **completion time** C_{ij}, C_j:
    time when job j execution on machine i is finished

Graham's classification $\alpha|\beta|\gamma$

(Many) Scheduling problems can be described by a three field notation

- α: the machine environment
  - describes a way of job assingments to machines
- β: the job characteristics,
  - describes constraints applied to jobs
- γ: the objective criterion to be minimized
- complexity for combinations of scheduling problems

Examples

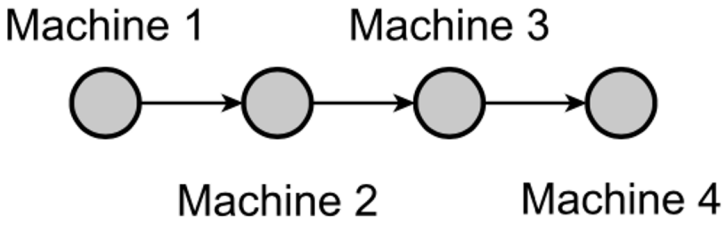
- $P3|prec|C_{max}$: bike assembly
- $Pm|r_j|\sum w_j C_j$: parallel machines

- **Single machine** ($\alpha = 1$): $1|\ldots|\ldots$
- **Identical parallel machines** Pm
  - m identical machines working in parallel with the same speed
  - each job consist of a single operation,
  - each job processed by any of the machines m for p_j time units
- **Uniform parallel machines** Qm
  - processing time of job j on machine i propotional to its speed v_i
  - $p_{ij} = p_j / v_i$
  - ex. several computers with processor different speed
- **Unrelated parallel machines** Rm
  - machine have different speed for different jobs
  - machine i process job j with speed v_{ij}
  - $p_{ij} = p_j / v_{ij}$
  - ex. vector computer computes vector tasks faster than a classical PC

- **Shop Problems**
  - each tasks is executed sequentially on several machine
    - job j consists of several operations (i, j)
    - operation (i, j) of jobs j is performed on machine i withing time p_{ij}
    - ex: job j with 4 operations $(1, j), (2, j), (3, j), (4, j)$

- Shop problems are classical
  studied in details in **operations research**
- Real problems are ofter more complicated
  - utilization of knowledge on subproblems or simplified problems in solutions

- **Shop Problems**
  - each tasks is executed sequentially on several machine
    - job j consists of several operations (i, j)
    - operation (i, j) of jobs j is performed on machine i withing time p_{ij}
    - ex: job j with 4 operations $(1, j), (2, j), (3, j), (4, j)$



- Shop problems are classical studied in details in **operations research**
- Real problems are ofter more complicated
  - utilization of knowledge on subproblems or simplified problems in solutions

- **Flow shop** *Fm*
  - m machines in series
  - each job has to be processed on each machine
  - all jobs follow the same route:
    - first machine 1, then machine 2, ...
  - if the jobs have to be processed in the same order on all machines, we have a **permutation** flow shop
- **Flexible flow shop** *FFs*
  - a generalizatin of flow shop problem
  - s phases, a set of parallel machines is assigned to each phase
  - i.e. flow shop with s parallel machines
  - each job has to be processed by all phase in the same order
    - first on a machine of phase 1, then on a machine of phase 2, ...
  - the task can be performed on any machine assigned to a given phase

- **Job shop** *Jm*
  - flow shop with m machines
  - each job has its individual predetermined route to follow
    - processing time of a given jobs might be zero for some machines
  - $(i,j) \rightarrow (k,j)$ specifies that job j is performed on machine i earlier than on machine k
  - example: $(2,j) \rightarrow (1,j) \rightarrow (3,j) \rightarrow (4,j)$
- **Open shop** *Om*
  - flow shop with m machines
  - processing time of a given jobs might be zero for some machines
  - no routing restrictions (this is a scheduling decision)

- **Precedence constraints** *prec*
  - linear sequence, tree structure
  - for jobs a, b we write $a \rightarrow b$, with meaning of $S_a + p_a \leqslant S_b$
  - example: bike assembly
- **Preemptions** *pmtn*
  - a job with a higher priority interrupts the current job
- **Machine suitability** M_j
  - a subset of machines M_j, on which job j can be executed
  - room assignment: appropriate size of the classroom
  - games: a computer with a HW graphical library
- **Work force constraints** W, W_ℓ
  - another sort of machines is introduced to the problem
  - machines need to be served by operators and jobs can be performed only if operators are available, operators W
  - different groups of operators with a specific qualification can exist, W_ℓ is a number of operators in group ℓ

- **Routing constraints**
  - determine on which machine jobs can be executed,
  - an order of job execution in shop problems
    - job shop problem: an operation order is given in advance
    - open shop problem: a route for the job is specified during scheduling
- **Setup time and cost** $s_{ijk}, c_{ijk}, s_{jk}, c_{jk}$
  - depend on execution sequence
  - s_{ijk} time for execution of job k after job j on machine i
  - c_{ijk} cost of execution of job k after job j on machine i
  - s_{jk}, c_{jk} time/cost independent on machine
  - examples
    - lemonade filling into bottles
    - travelling salesman problem $1|s_{jk}|C_{max}$

- **Makespan** C_{max}: maximum completion time

$$C_{max} = max(C_1, \ldots, C_n)$$

- Example: $C_{max} = max\{1, 3, 4, 5, 8, 7, 9\} = 9$

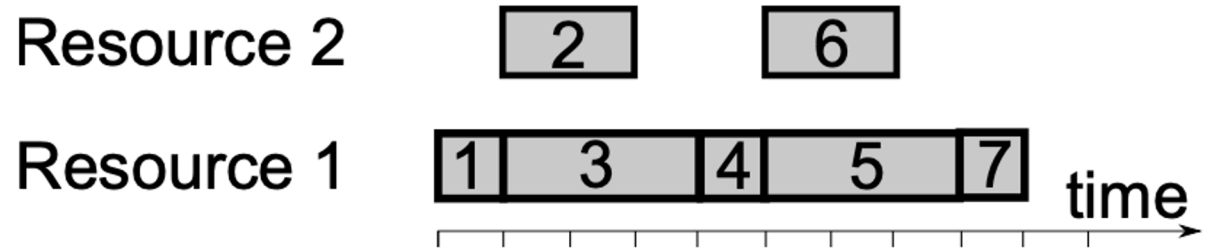
- Goal: **makespan minimization** often
  - maximizes **throughput**
  - ensures **uniform load of machines (***load balancing***)**
  - example: $C_{max} = max\{1, 2, 4, 5, 7, 4, 6\} = 7$

  - It is a basic criterion that is used very often.

- **Makespan** C_{max}: maximum completion time

$$C_{max} = max(C_1, \ldots, C_n)$$

- Example: $C_{max} = max\{1, 3, 4, 5, 8, 7, 9\} = 9$



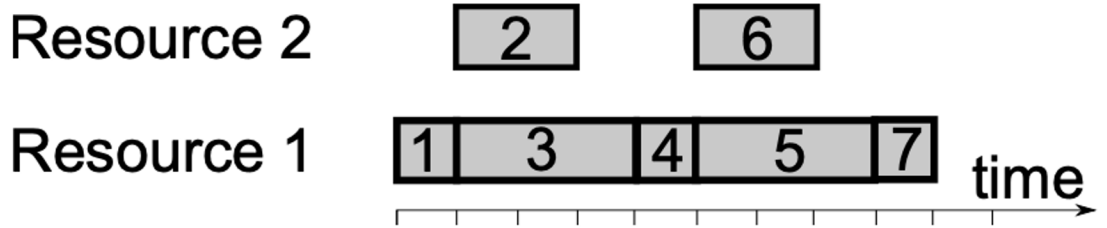
- Goal: **makespan minimization** often
  - maximizes **throughput**
  - ensures **uniform load of machines (*load balancing*)**
  - example: $C_{max} = max\{1, 2, 4, 5, 7, 4, 6\} = 7$

  - It is a basic criterion that is used very often.

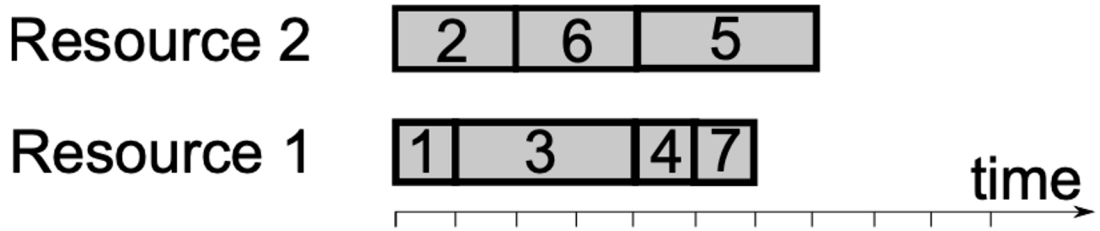
- **Makespan** C_{max}: maximum completion time

$$C_{max} = max(C_1, \ldots, C_n)$$

- Example: $C_{max} = max\{1, 3, 4, 5, 8, 7, 9\} = 9$



- Goal: **makespan minimization** often
  - maximizes **throughput**
  - ensures **uniform load of machines (***load balancing***)**
  - example: $C_{max} = max\{1, 2, 4, 5, 7, 4, 6\} = 7$



  - It is a basic criterion that is used very often.

- **Lateness** of job j: $L_{max} = C_j - d_j$
- **Maximum lateness** L_{max}

$$L_{max} = max(L_1, \ldots, L_n)$$

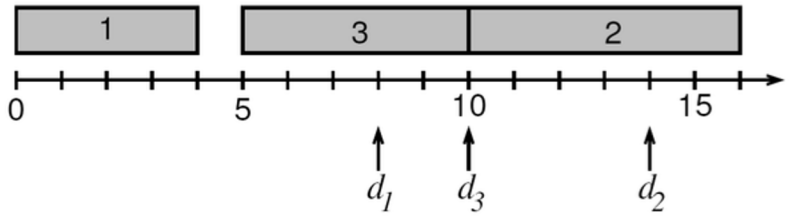
- Goal: **maximum lateness minimization**
- Example:

$$\begin{aligned} L_{max} &= max(L_1, L_2, L_3) = \\ &= max(C_1 - d_1, C_2 - d_2, C_3 - d_3) = \\ &= max(4 - 8, 16 - 14, 10 - 10) = \\ &= max(-4, 2, 0) = 2 \end{aligned}$$

- **Lateness** of job j: $L_{max} = C_j - d_j$
- **Maximum lateness** L_{max}

$$L_{max} = max(L_1, \ldots, L_n)$$

- Goal: **maximum lateness minimization**
- Example:



$$\begin{aligned} L_{max} &= max(L_1, L_2, L_3) = \\ &= max(C_1 - d_1, C_2 - d_2, C_3 - d_3) = \\ &= max(4 - 8, 16 - 14, 10 - 10) = \\ &= max(-4, 2, 0) = 2 \end{aligned}$$

- **Job tardiness** j: $T_j = max(C_j - d_j, 0)$
- **Total tardiness**

$$\sum_{j=1}^{n} T_j$$

- Goal: **total tardiness minimization**
- Example: $T_1 + T_2 + T_3 =$

$$\begin{aligned} &= \max(C_1 - d_1, 0) + \max(C_2 - d_2, 0) + \max(C_3 - d_3, 0) = \\ &= \max(4 - 8, 0) + \max(16 - 14, 0) + \max(10 - 10, 0) = \\ &= 0 + 2 + 0 = 2 \end{aligned}$$

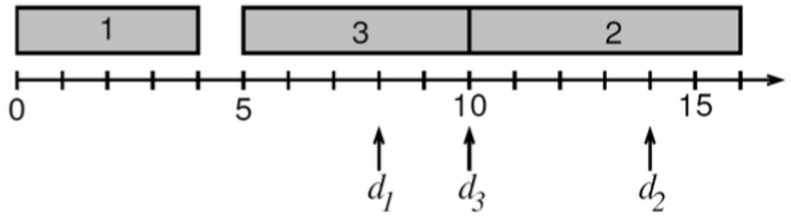
- **Total weighted tardiness**

$$\sum_{j=1}^{n} w_j T_j$$

- Goal: **total weighted tardiness minimization**

- **Job tardiness j:** $T_j = max(C_j - d_j, 0)$
- **Total tardiness**

$$\sum_{j=1}^{n} T_j$$



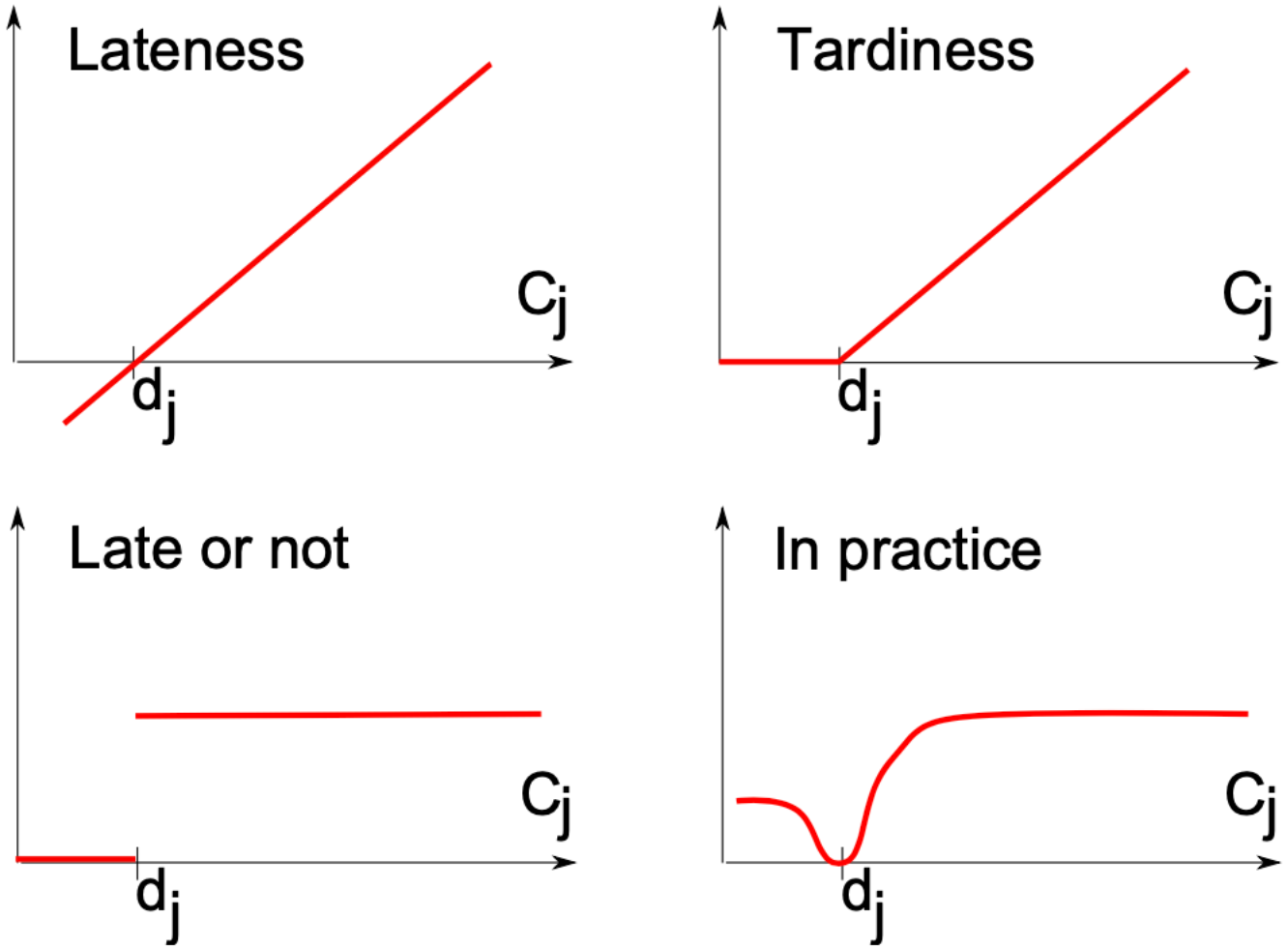
- Goal: **total tardiness minimization**
- Example: $T_1 + T_2 + T_3 =$

$$\begin{aligned} &= \max(C_1 - d_1, 0) + \max(C_2 - d_2, 0) + \max(C_3 - d_3, 0) = \\ &= \max(4 - 8, 0) + \max(16 - 14, 0) + \max(10 - 10, 0) = \\ &= 0 + 2 + 0 = 2 \end{aligned}$$

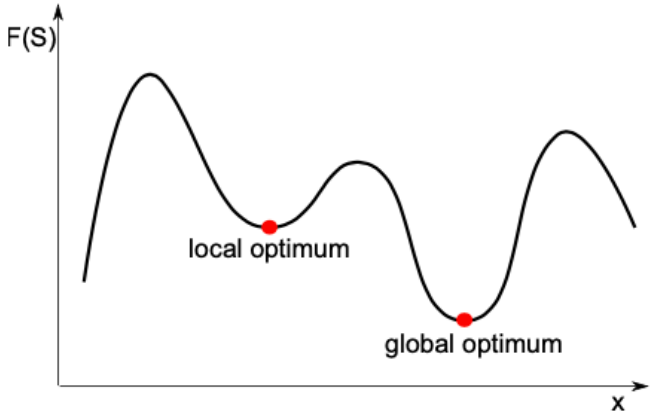
- **Total weighted tardiness**

$$\sum_{j=1}^{n} w_j T_j$$

- Goal: **total weighted tardiness minimization**


Lateness
C_j
d_j
Tardiness
C_j
d_j
Late or not
C_j
d_j
In practice
C_j
d_j

- **Constructive methods**
  - Start with the empty schedule
  - Add step by step other jobs to the schedule so that the schedule remains consistent
- **Local search**
  - Start with a complete non-consistent schedule
    - trivial: random generated
  - Try to find a better "similar" schedule by local modifications.
  - Schedule quality is evaluated using optimization criteria
    - ex. makespan
  - optimization criteria assess also schedule consistency
    - ex. a number of vialoted precedence constraints
  - **Hybrid approaches**
    - combinations of both methods



1. **Initialization**
   - $k = 0$
   - Select an initial schedule S_0
   - Record the current best schedule: $S_{best} = S_0$ a $cost_{best} = F(S_0)$
2. **Select and update**
   - Select a schedule from neighborhood: $S_{k+1} \in N(S_k)$
   - if no element $N(S_k)$ satisfies schedule acceptance criterion then the algorithms finishes
   - if $F(S_{k+1}) < cost_{best}$ then $S_{best} = S_{k+1}$ a $cost_{best} = F(S_{k+1})$
3. **Finish**
   - if the stop constraints are satisfied then the algorithms finishes
   - otherwise $k = k + 1$ and continue with step 2.

- **Schedule representation**
  - permutations n jobs
  - example with six jobs: $1, 4, 2, 6, 3, 5$
- **Neighborhood definition**
  - pairwise exchange of neighboring jobs
    - $n-1$ possible schedules in the neighborhood
    - example: $1, 4, 2, 6, 3, 5$ is modified to $1, 4, 2, 6, 5, 3$
  - or select an arbitrary job from the schedule and place it to an arbitrary position
    - $\leqslant n(n-1)$ possible schedules in the neighborhood
    - example: from $1, 4, 2, 6, 3, 5$ we select randomly 4 and place it somewhere else: $1, 2, 6, 3, 4, 5$

- Criteria for schedule selection
  - **Criterion for schedule acceptance/refuse**
- The main difference among a majority of methods
  - to accept a better schedule all the time?
  - to accept even worse schedule sometimes?
- methods
  - probabilistic
    - **random walk**: with a small probability (ex. 0.01) a worse schedule is accepted
    - **simulated annealing**
  - deterministic
    - **tabu search**: a tabu list of several last state/modifications that are not allowed for the following selection is maintained

TABU SCHEDULING

- **Deterministic criterion for schedule acceptance/refuse**
- **Tabu list** of several last schedule modifications is maintained
  - each new modification is stored on the top of the tabu list
    - ex. of a store modification: exchange of jobs j and k
  - tabu list = a list of forbidden modifications
  - the neighborhood is constrained over schedules, that do not require a change in the tabu list
    - a protection against cycling
    - example of a trivial cycling:
      the first step: exchange jobs 3 and 4, the second step: exchange jobs 4 and 3
  - a fixed length of the list (often: 5-9)
    - the oldest modifications of the tabu list are removed
    - too small length: cycling risk increases
    - too high length: search can be too constrained
- **Aspiration criterion**
  - determines when it is possible to make changes in the tabu list
  - ex. a change in the tabu list is allowed if $F(S_{best})$ is improved.

1.
   - $k = 1$
   - Select an initial schedule S_1 using a heuristics,
     $S_{best} = S_1$
2.
   - Choose $S_c \in N(S_k)$
   - If the modification $S_k \rightarrow S_c$ is forbidden because it is in the tabu list
     then continue with step 2
3.
   - If the modification $S_k \rightarrow S_c$ is not forbidden by the tabu list
     then $S_{k+1} = S_c$,
     store the reverse change to the top of the tabu list
     move other positions in the tabu list one position lower
     remove the last item of the tabu list
   - if $F(S_c) < F(S_{best})$ then $S_{best} = S_c$
4.
   - $k = k + 1$
   - if a stopping condition is satisfied then finish
     otherwise continue with step 2.

A schedule problem with $1|d_j|\sum w_j T_j$

- remind: $T_j = \max(C_j - d_j, 0)$

jobs	1	2	3	4
p_j	10	10	13	4
d_j	4	2	1	12
w_j	14	12	1	12

- Neighborhood: all schedules obtained by pair exchange of neighbor jobs
- Schedule selection from the neighborhood: select the best schedule
- Tabu list: pairs of jobs (j, k) that were exchanged in the last two modifications
- Apply tabu search for the initial solution $(2, 1, 4, 3)$
- Perform four iterations

TABU SEARCH & 1-MACHINE

Example min $\sum w_j T_j$

Tabu List Size = 2

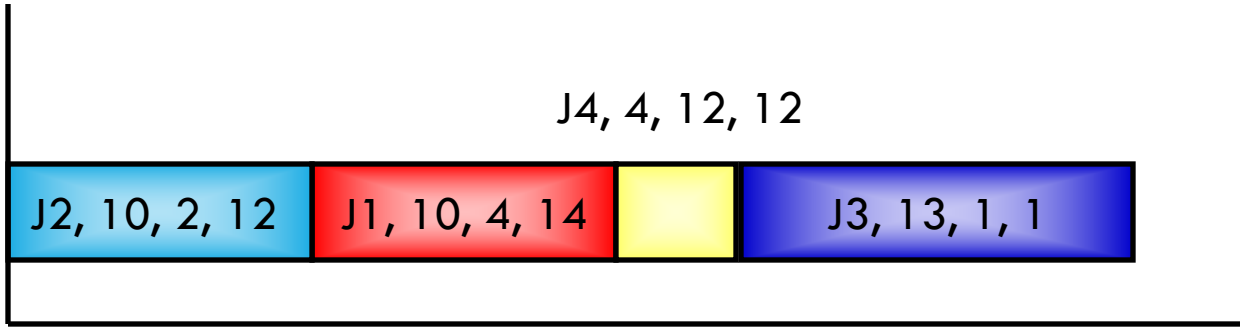
Jobs	1	2	3	4
p_j	10	10	13	4
d_j	4	2	1	12
w_j	14	12	1	12

J_i, p_i, d_i, w_i

J1, 10, 4, 14

TS STEP 1: FIND INITIAL SOLUTION

Arbitrarily choose $(2, 1, 4, 3)$

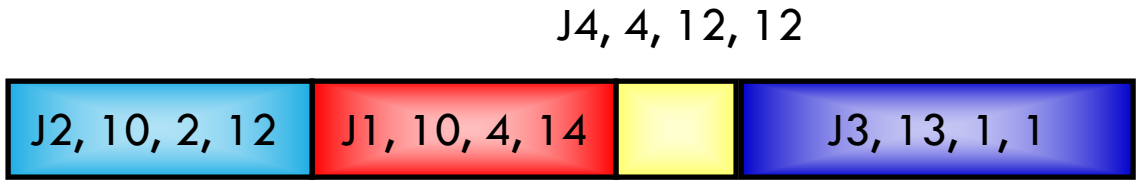


$$\sum w_j T_j = 8 \times 12 + 16 \times 14 + 12 \times 12 + 36 \times 1 = 500$$

TS STEP 2: EVALUATE NEIGHBORHOOD & SELECT MOVE

Neighborhood: swap

- Adjacent pairwise interchange



Choose (1,4)

J4, 4, 12, 12

J1, 10, 4, 14	J2, 10, 2, 12		J3, 13, 1, 1

$\sum w_j T_j = 480$

J4, 4, 12, 12

J2, 10, 2, 12		J1, 10, 4, 14	J3, 13, 1, 1

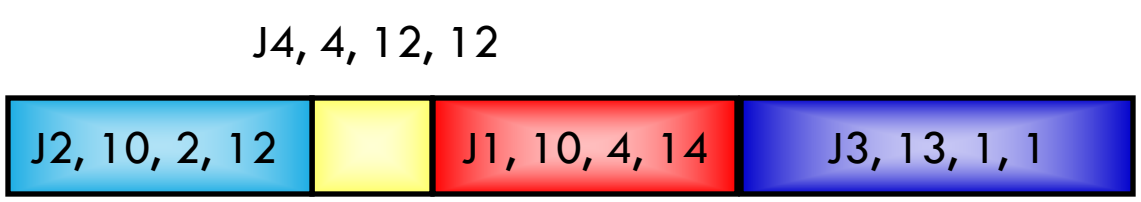
$\sum w_j T_j = 436$

J4, 4, 12, 12

J2, 10, 2, 12	J1, 10, 4, 14	J3, 13, 1, 1	

$\sum w_j T_j = 652$

TS STEP 3: ADD MOVE TO TABU LIST



$\sum w_j T_j = 436$

Tabu: ((1, 4))

Best so far:
(2, 4, 1, 3): 436

TS STEP 2: EVALUATE NEIGHBORHOOD & SELECT MOVE

J4, 4, 12, 12

$\sum w_j T_j = 436$

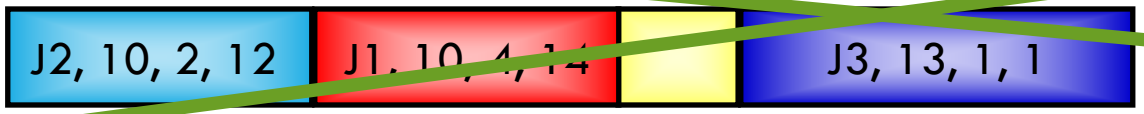
Tabu: ((1, 4))

Choose (4,2)

J4, 4, 12, 12

$\sum w_j T_j = 460$

J4, 4, 12, 12



$\sum w_j T_j = 500$

J4, 4, 12, 12

$\sum w_j T_j = 608$

TS STEP 3: ADD MOVE TO TABU LIST

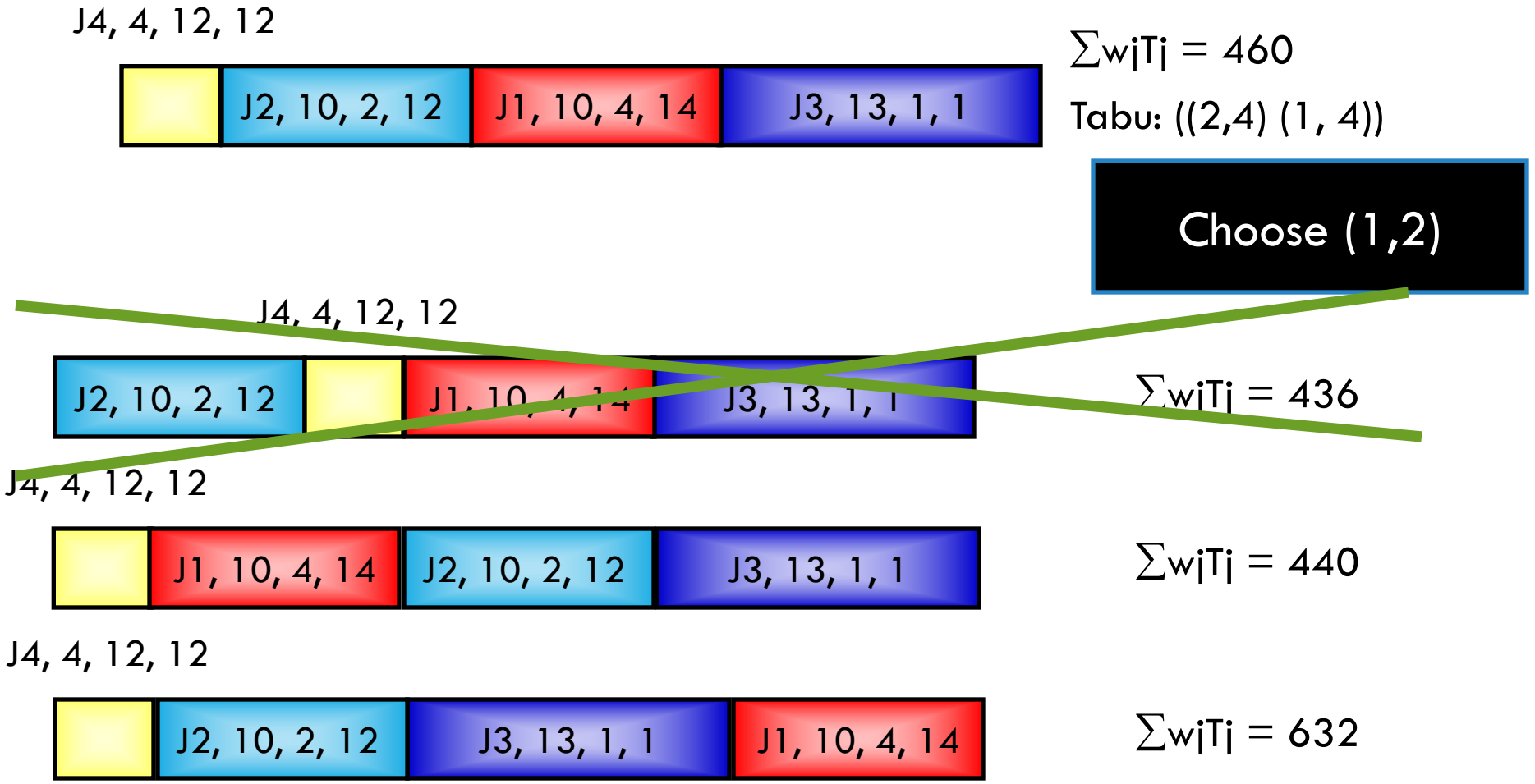
J4, 4, 12, 12

$\sum w_j T_j = 460$

Tabu: ((2,4) (1, 4))

Best so far:
(2, 4, 1, 3): 436

TS STEP 2: EVALUATE NEIGHBORHOOD & SELECT MOVE



TS STEP 3: ADD MOVE TO TABU LIST

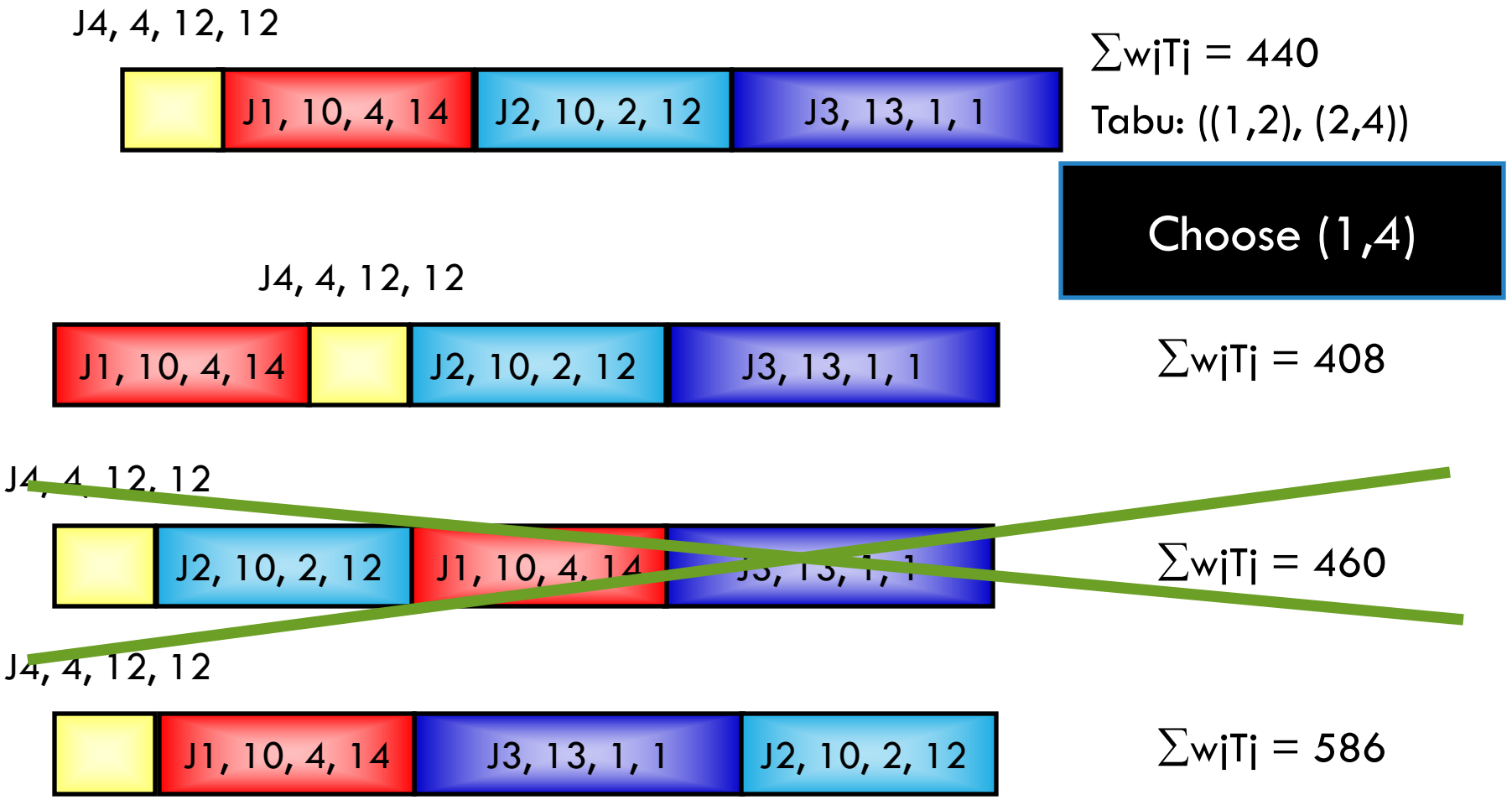
J4, 4, 12, 12

$\sum w_j T_j = 440$

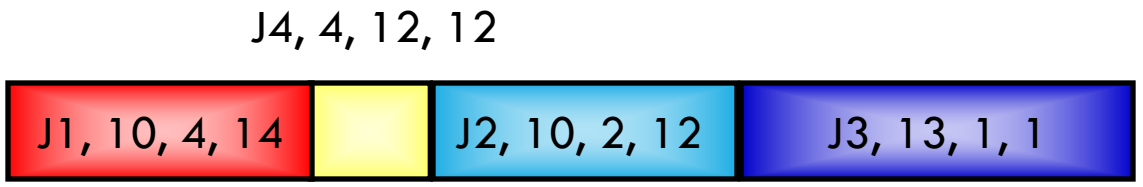
Tabu: ((1,2), (2,4))

Best so far:
(2, 4, 1, 3): 436

TS STEP 2: EVALUATE NEIGHBORHOOD & SELECT MOVE



TS STEP 3: ADD MOVE TO TABU LIST



$\sum w_j T_j = 408$

Tabu: ((1,4), (1,2))

Best so far:
(1,4,2,3): 408

And so on until a bound on the number of iterations

- (actually 408 is optimal, but TS has no way of knowing)

CRYSTAL MAZE

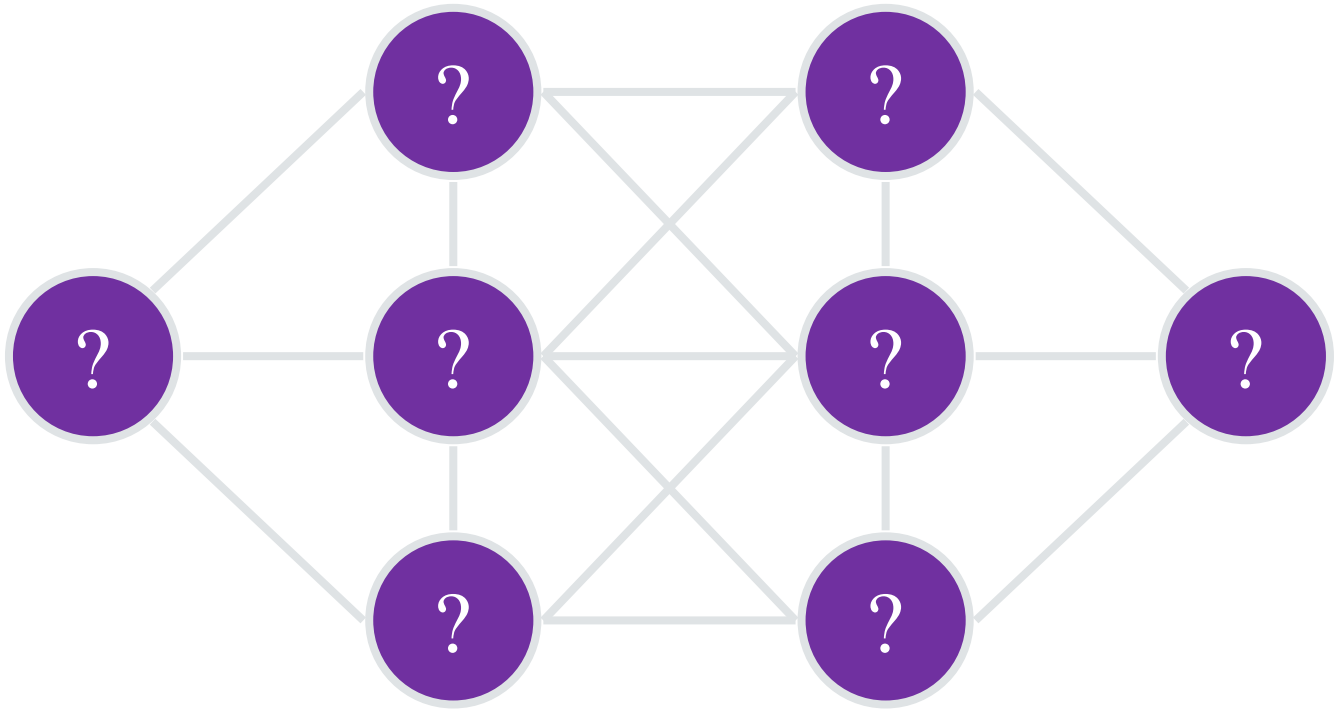
from

CRYSTAL MAZE

Place the numbers 1 through 8 in the nodes such that:

- Each number appears exactly once

  – No connected nodes have consecutive numbers



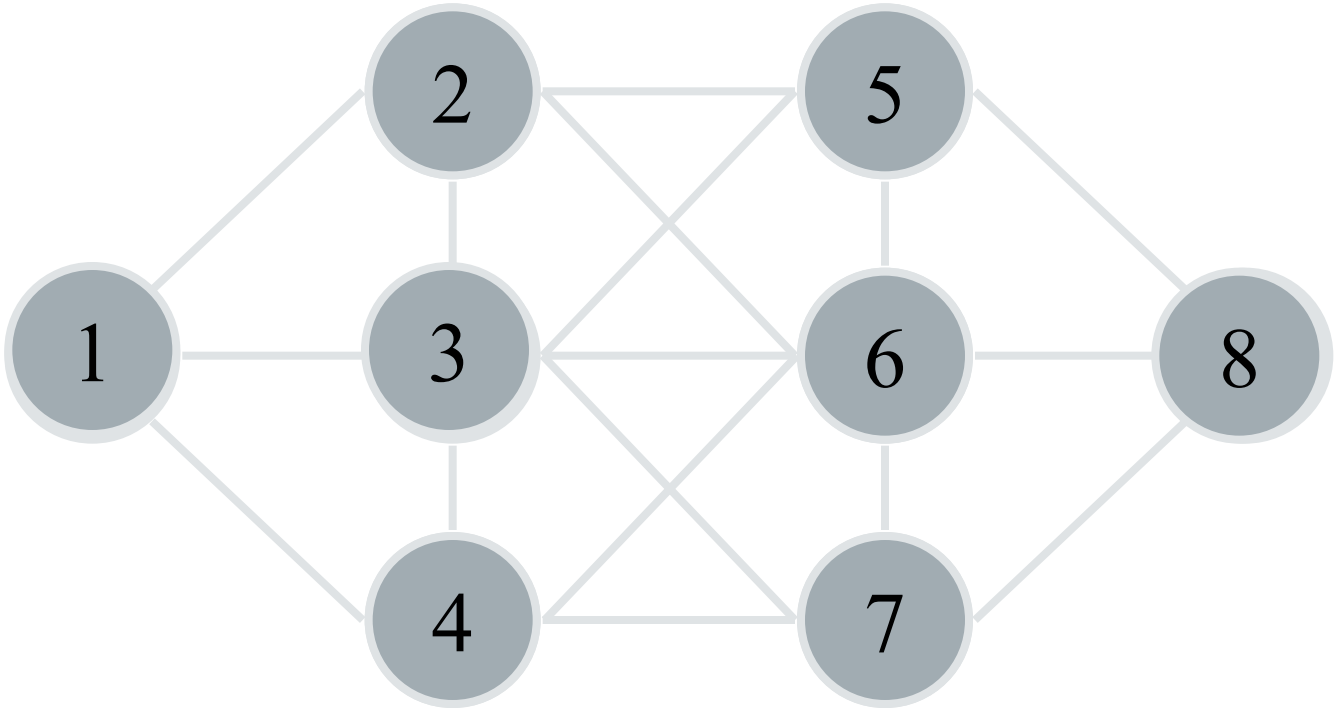
LOCAL SEARCH IDEA

Randomly assign values (even if the constraints are “broken”)

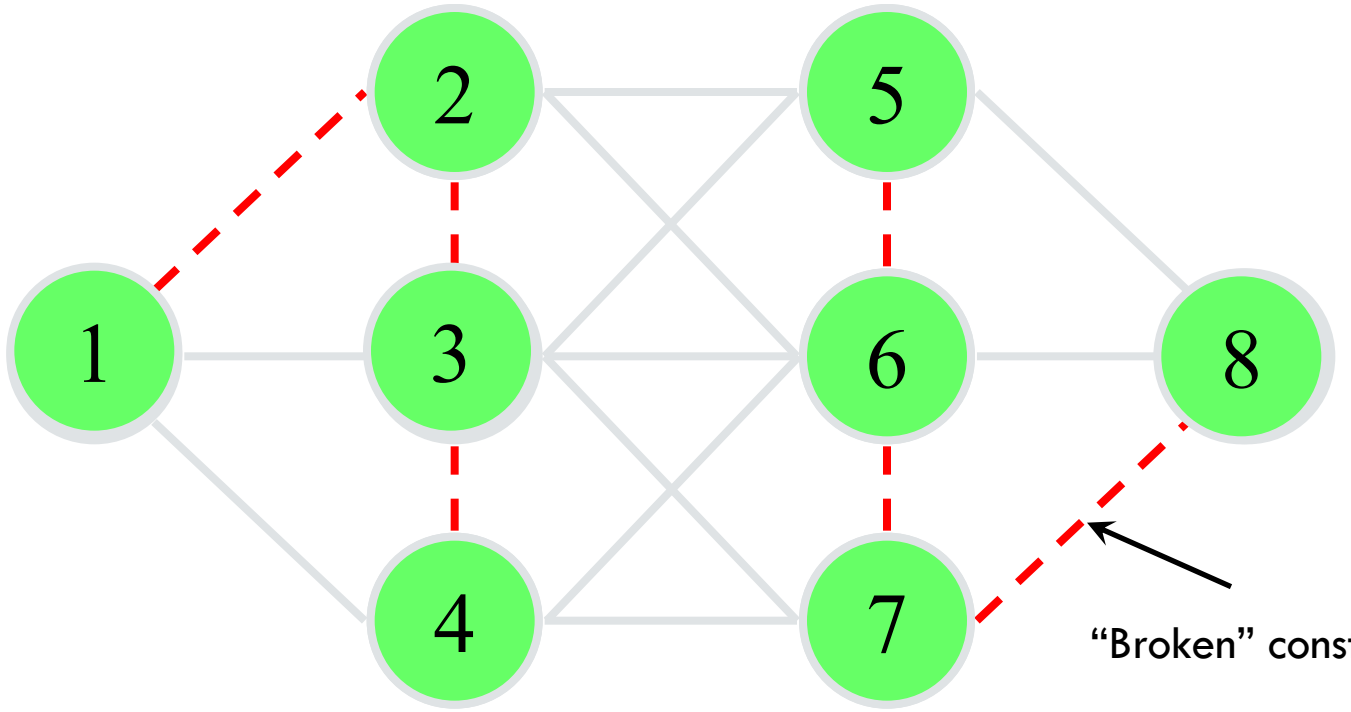
- Initial state will probably be infeasible

Make “moves” to try to move toward a solution

RANDOM INITIAL SOLUTION



RANDOM INITIAL SOLUTION



Cost = # of broken constraints

WHAT SHOULD WE DO NOW?

Move:

- Swap two numbers

Which two numbers?

- Randomly pick a pair
- The pair that will lead to the biggest decrease in cost
  - Cost: number of broken constraints

WHAT SHOULD WE DO NOW?

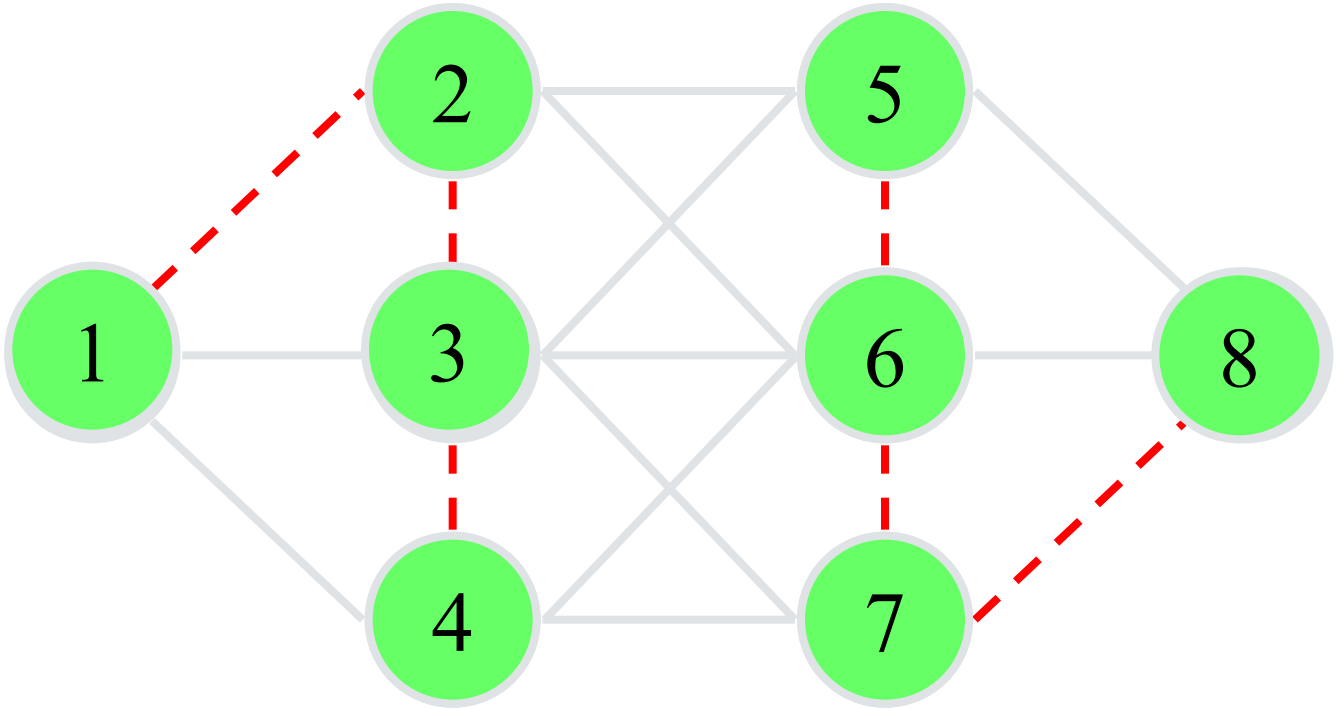
Move:

- Swap two numbers

Which two numbers?

- Randomly pick a pair
- The pair that will lead to the biggest decrease in cost
  - Cost: number of broken constraints

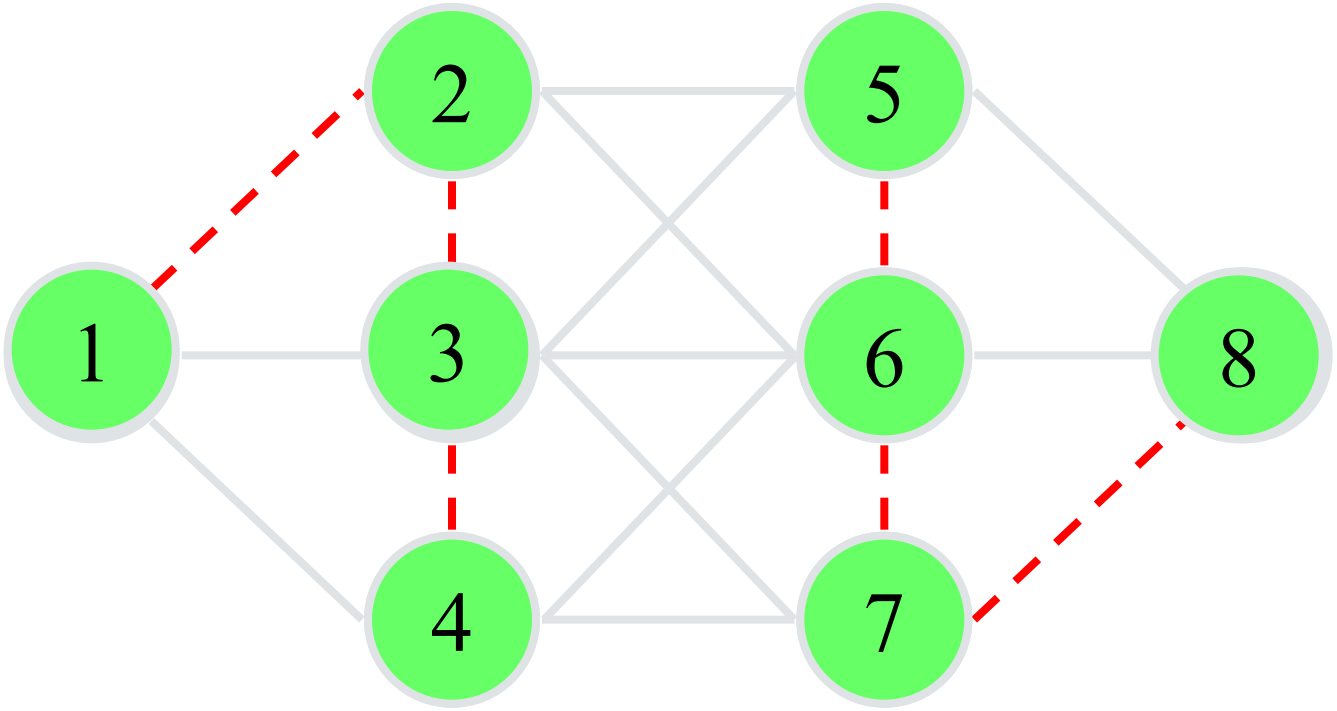
RANDOM INITIAL SOLUTION



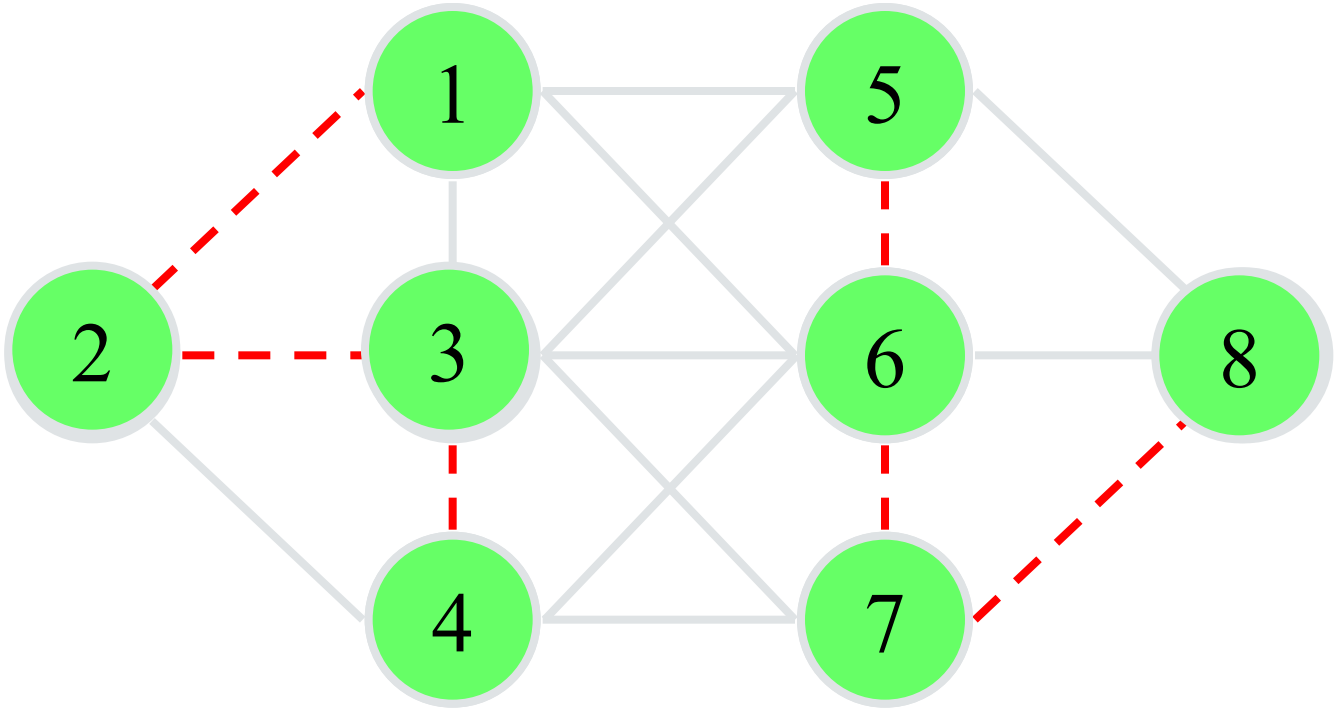
COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0							
2		0						
3			0					
4				0				
5					0			
6						0		
7							0	
8								0

RANDOM INITIAL SOLUTION



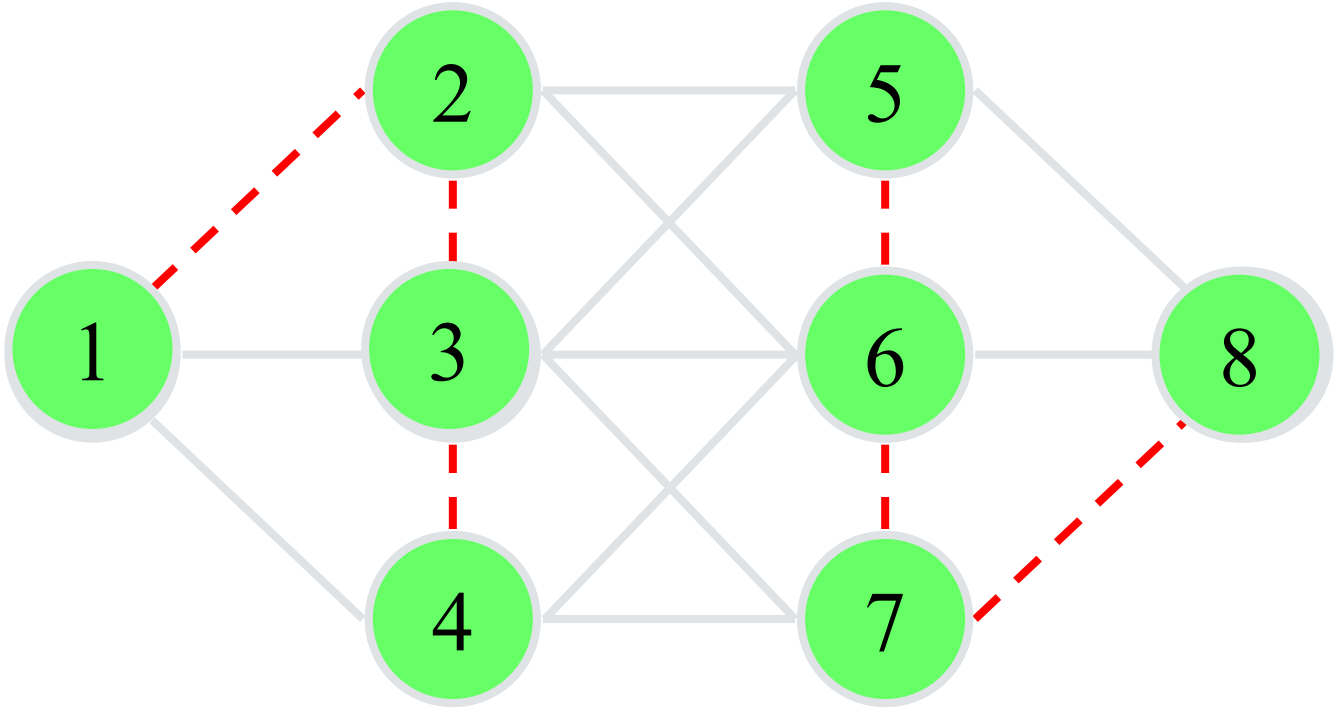
SWAP 1 & 2



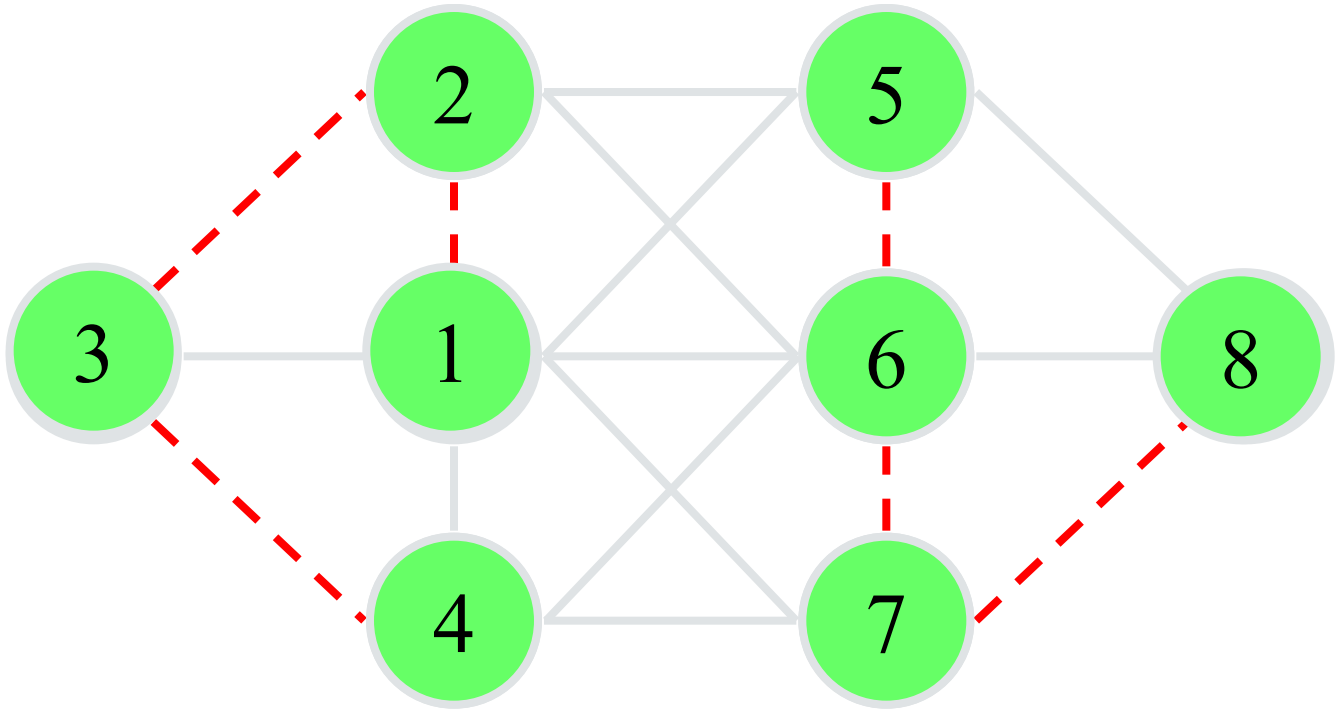
COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	0						
2		0						
3			0					
4				0				
5					0			
6						0		
7							0	
8								0

RANDOM INITIAL SOLUTION



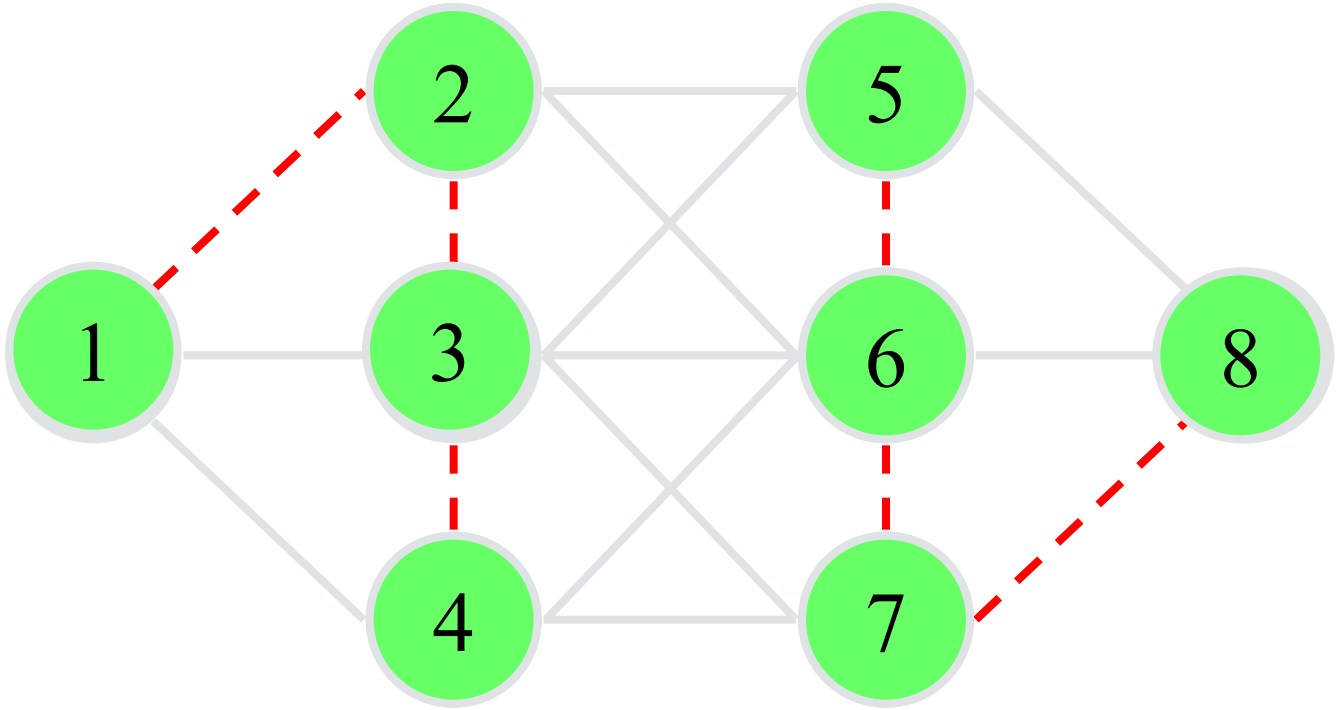
SWAP 1 & 3



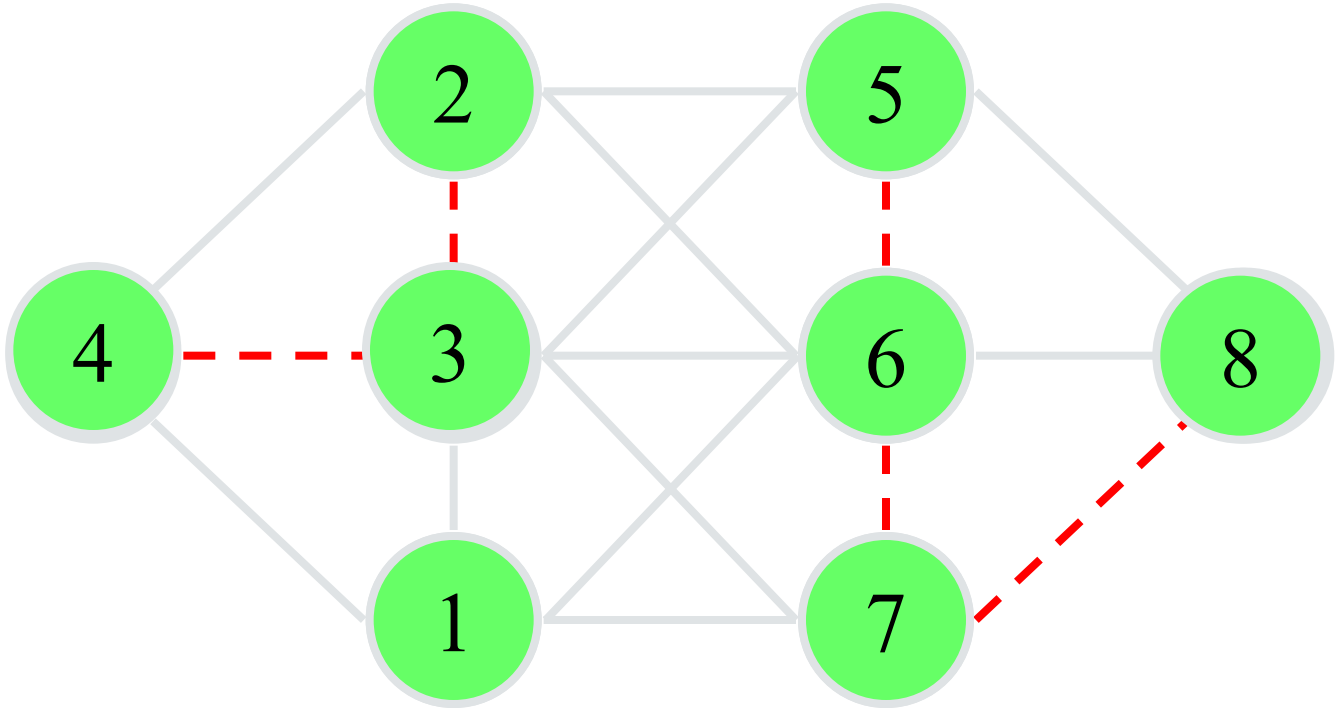
COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	0	0					
2		0						
3			0					
4				0				
5					0			
6						0		
7							0	
8								0

RANDOM INITIAL SOLUTION



SWAP 1 & 4



COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	0	0	-1				
2		0						
3			0					
4				0				
5					0			
6						0		
7							0	
8								0

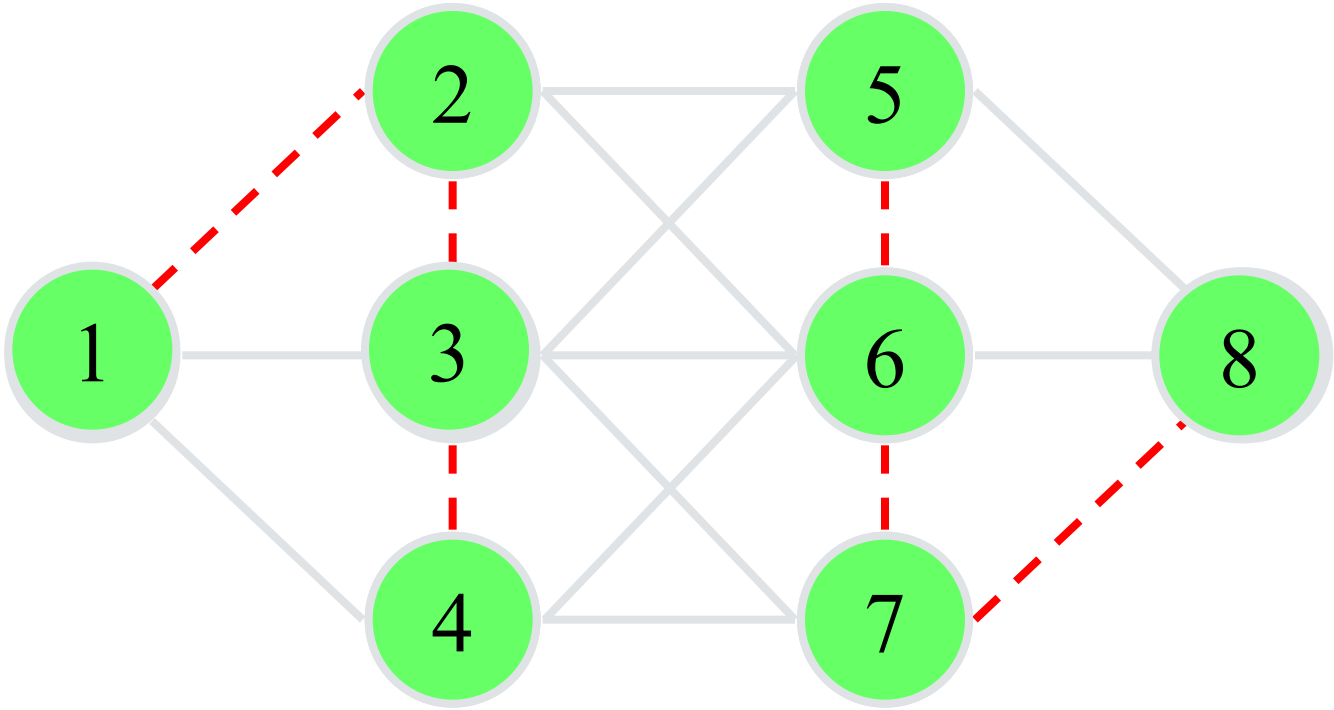
COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	0	0	-1	0	-2	-3	-2
2		0	-1	1	-1	-2	-1	-3
3			0	0	0	0	-1	0
4				0	0	0	-1	0
5					0	0	1	-1
6						0	-1	0
7							0	0
8								0

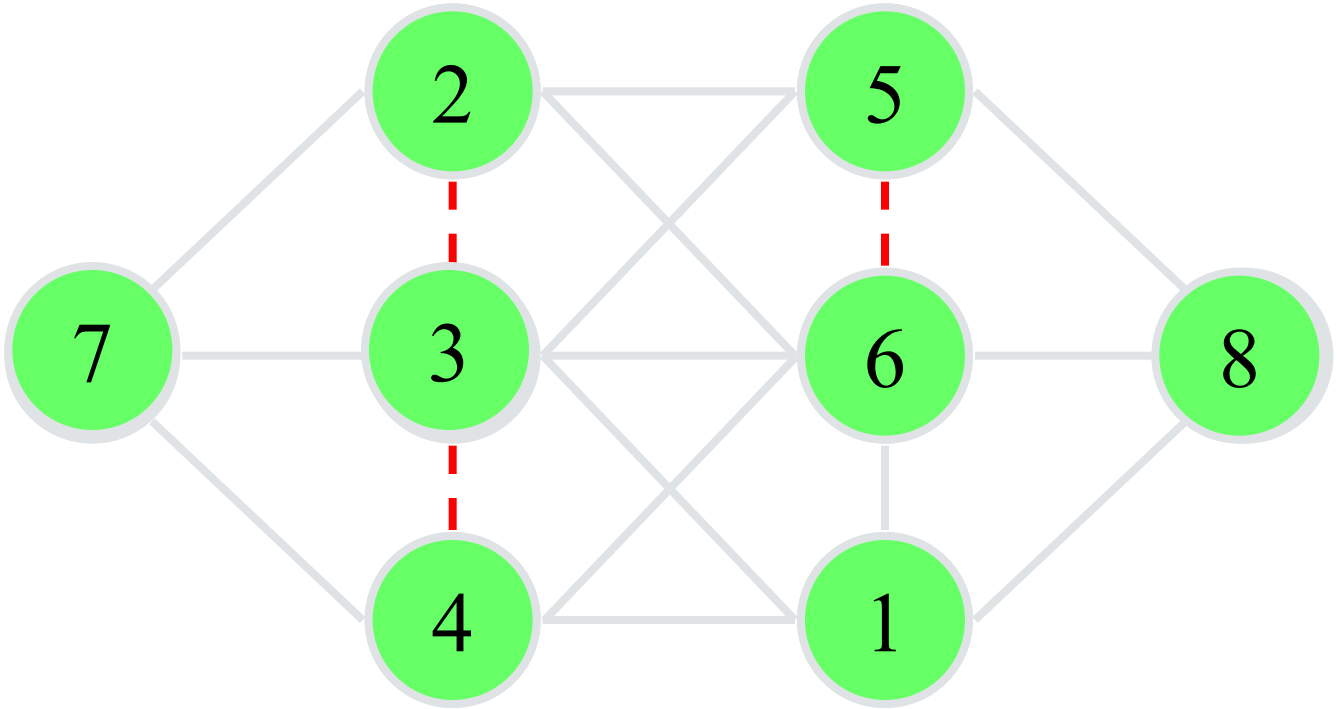
COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	0	0	-1	0	-2	-3	-2
2		0	-1	1	-1	-2	-1	-3
3			0	0	0	0	-1	0
4				0	0	0	-1	0
5					0	0	1	-1
6						0	-1	0
7							0	0
8								0

CURRENT STATE



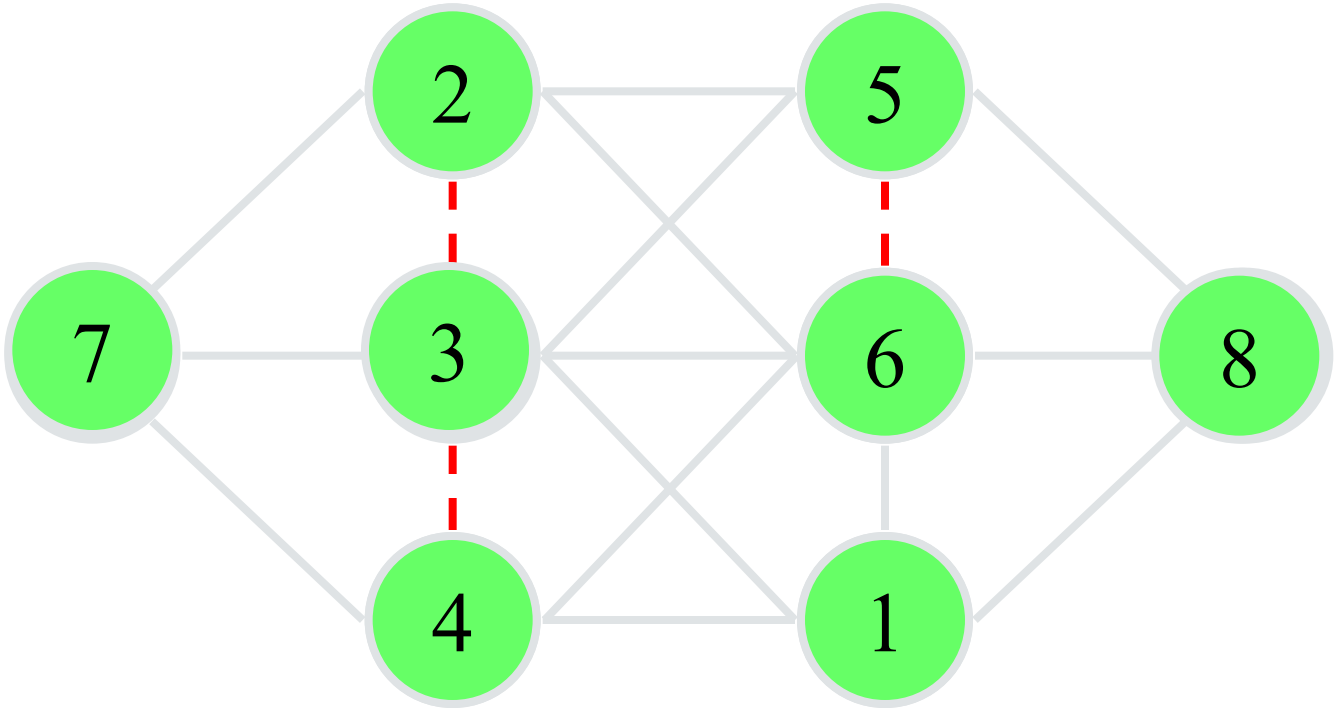
SWAP 1 & 7: COST 3



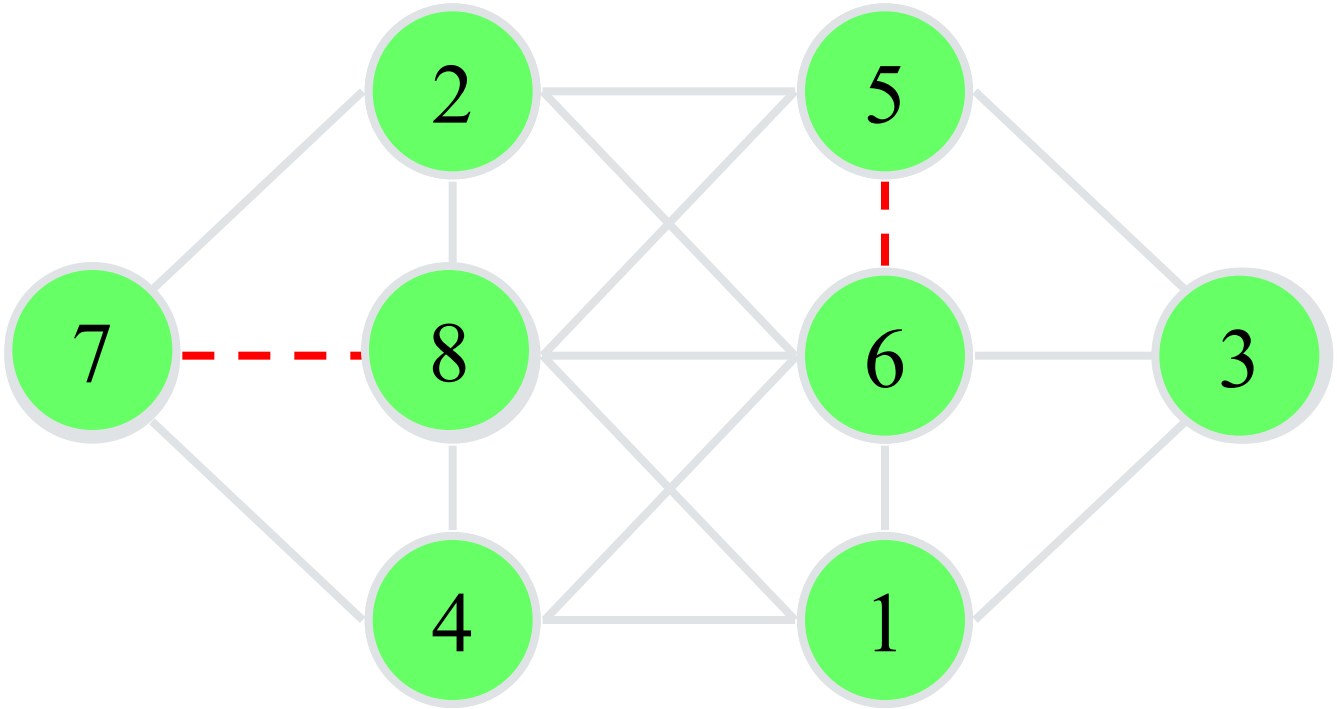
NEW COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	0	0	0	2	0	3	0
2		0	0	2	0	1	1	1
3			0	0	0	1	1	-1
4				0	0	1	1	1
5					0	1	2	0
6						0	0	0
7							0	1
8								0

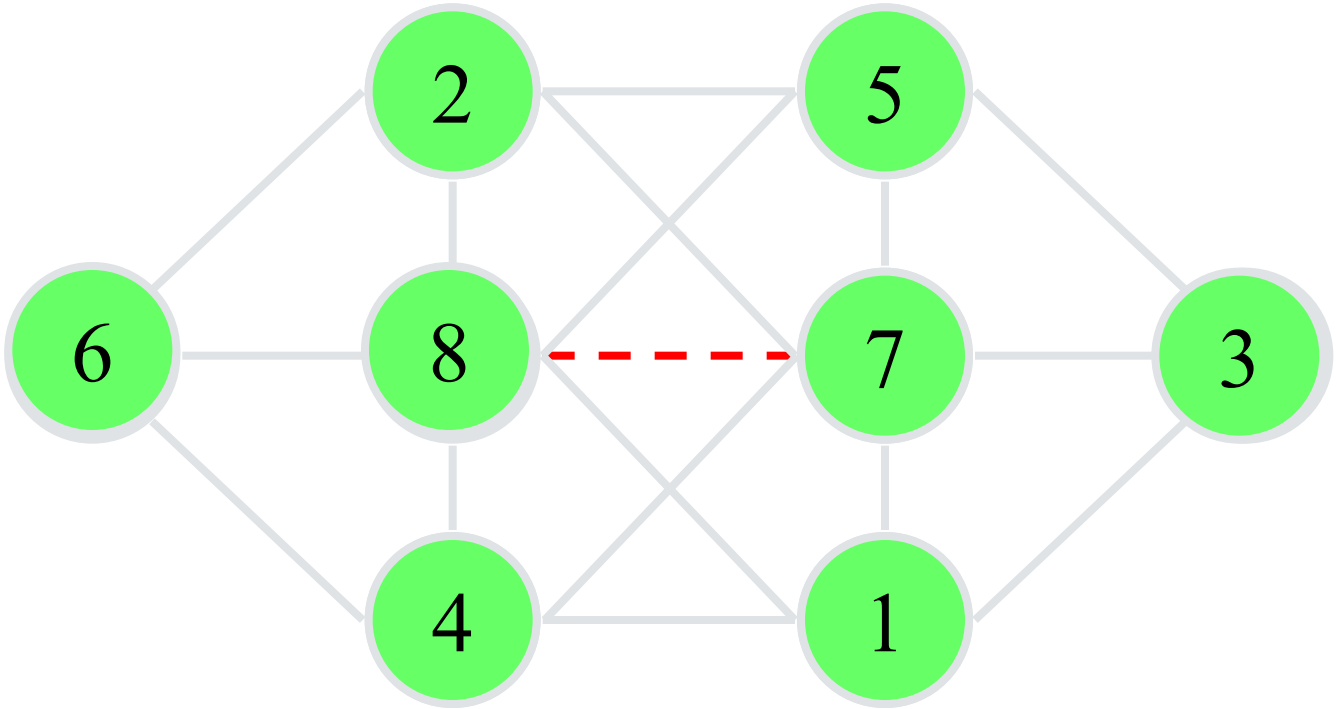
CURRENT STATE



SWAP 3 & 8: COST 2



SWAP 6 & 7: COST 1



MOVES

Initial State: Cost 6

Swap 1 & 7: Cost 3

Swap 3 & 8: Cost 2

Swap 6 & 7: Cost 1

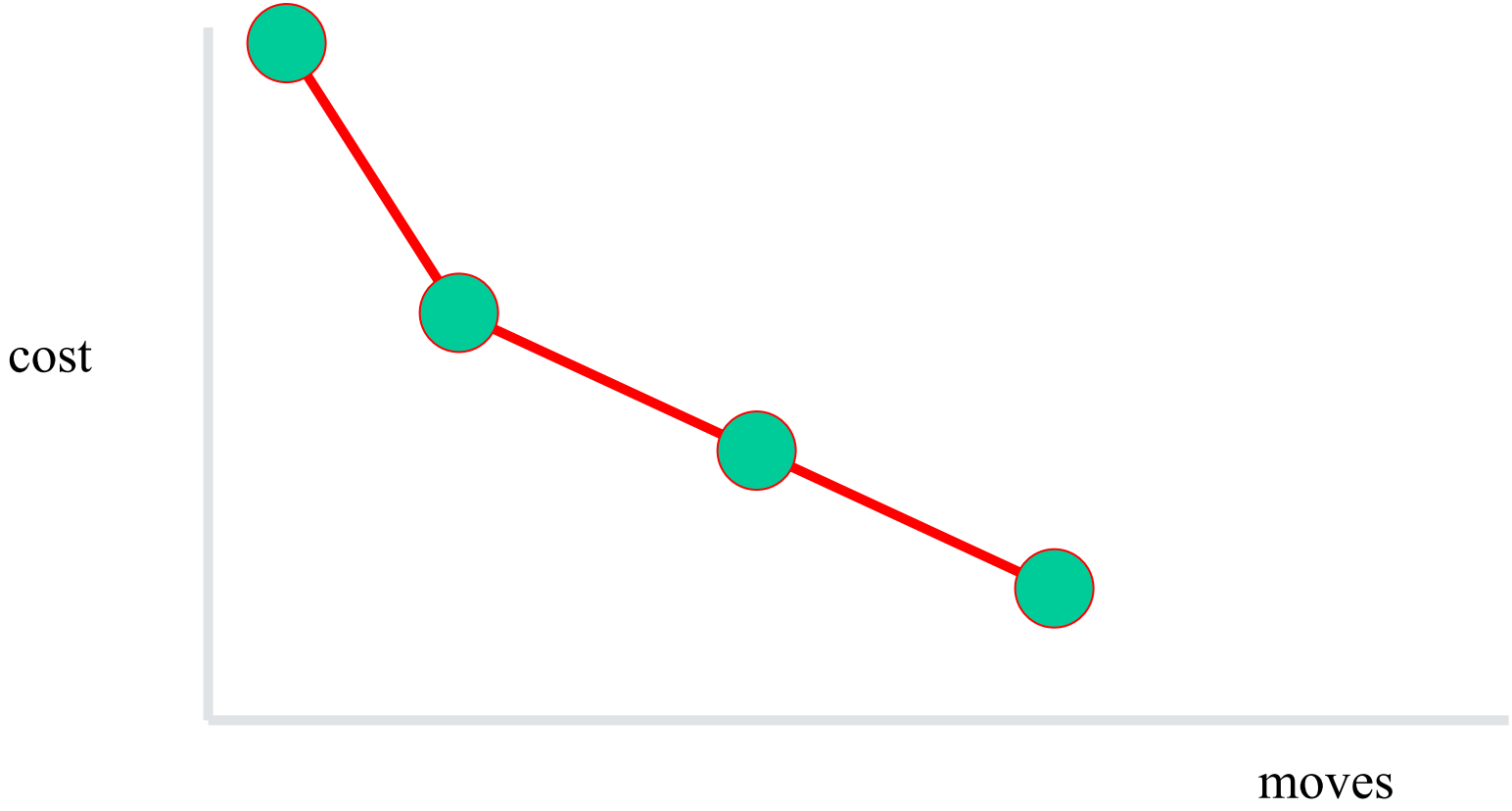
COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	1	1	1	2	2	1	1
2		0	1	2	2	1	3	1
3			0	1	1	4	1	2
4				0	2	1	3	1
5					0	2	1	2
6						0	1	1
7							0	1
8								0

NOW WHAT?

There are no improving moves to make!

So far, we have been “hill-climbing”



NOW WHAT?

Options:

- Restart from a new random state
- Take the least worse move (increase cost by minimal amount)
- Try a new style of local search

NOW WHAT?

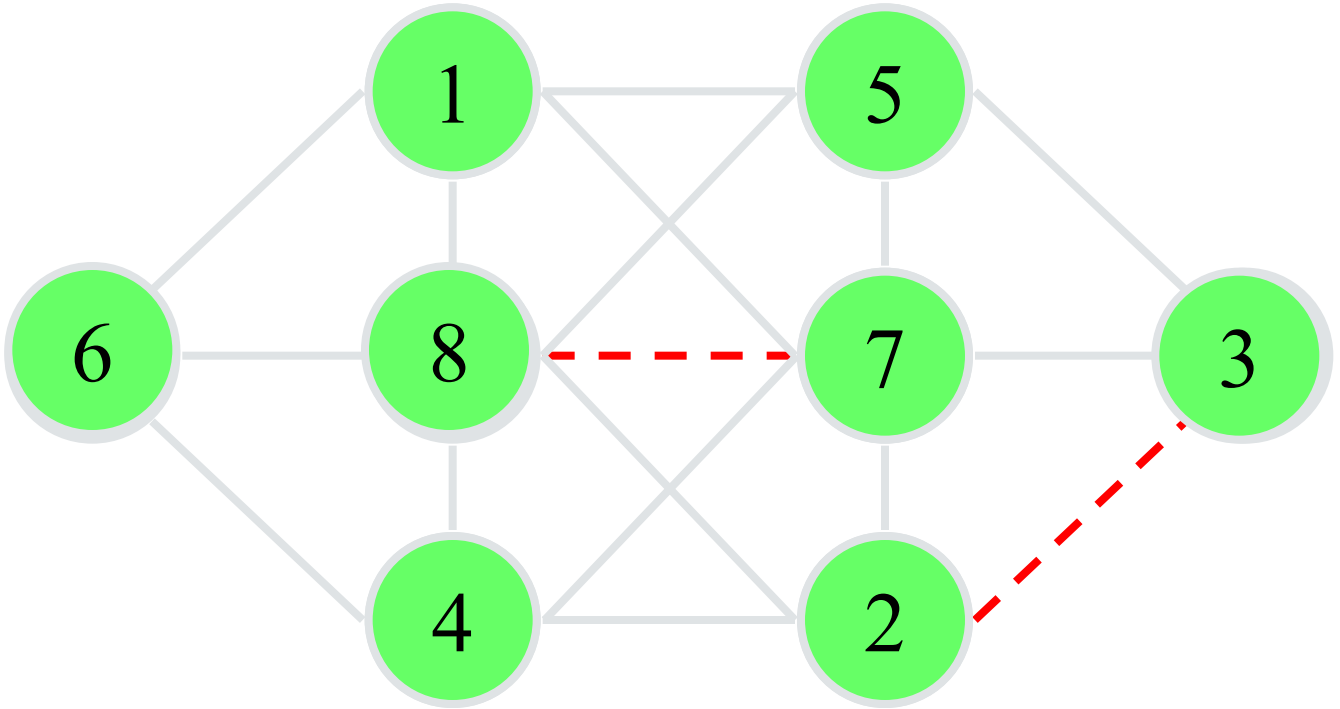
Options:

- Restart from a new random state
- Take the least worse move (increase cost by minimal amount)
- Try a new style of local search

COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	1	1	1	2	2	1	1
2		0	1	2	2	1	3	1
3			0	1	1	4	1	2
4				0	2	1	3	1
5					0	2	1	2
6						0	1	1
7							0	1
8								0

SWAP 1 & 2: COST 2



COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	-1	0	2	1	2	2	1
2		0	1	0	2	1	1	0
3			0	1	-1	2	0	1
4				0	2	1	3	1
5					0	1	1	2
6						0	1	1
7							0	1
8								0

COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	-1	0	2	1	2	2	1
2		0	1	0	2	1	1	0
3			0	1	-1	2	0	1
4				0	2	1	3	1
5					0	1	1	2
6						0	1	1
7							0	1
8								0

MOVES

Initial State: Cost 6

Swap 1 & 7: Cost 3

Swap 3 & 8: Cost 2

Swap 6 & 7: Cost 1

Swap 1 & 2: Cost 2

MOVES

Initial State: Cost 6

Swap 1 & 7: Cost 3

Swap 3 & 8: Cost 2

Swap 6 & 7: Cost 1

Swap 1 & 2: Cost 2

Swap 1 & 2: Cost 1

MOVES

Initial State: Cost 6

Swap 1 & 7: Cost 3

Swap 3 & 8: Cost 2

Swap 6 & 7: Cost 1

Swap 1 & 2: Cost 2

Swap 1 & 2: Cost 1

Swap 1 & 2: Cost 2

Swap 1 & 2: Cost 1 … and so on

NOW WHAT?

Options:

- Take the least worse move (increase cost by minimal amount)
- Restart from a new random state
- Try a new style of local search

NOW WHAT?

Options:

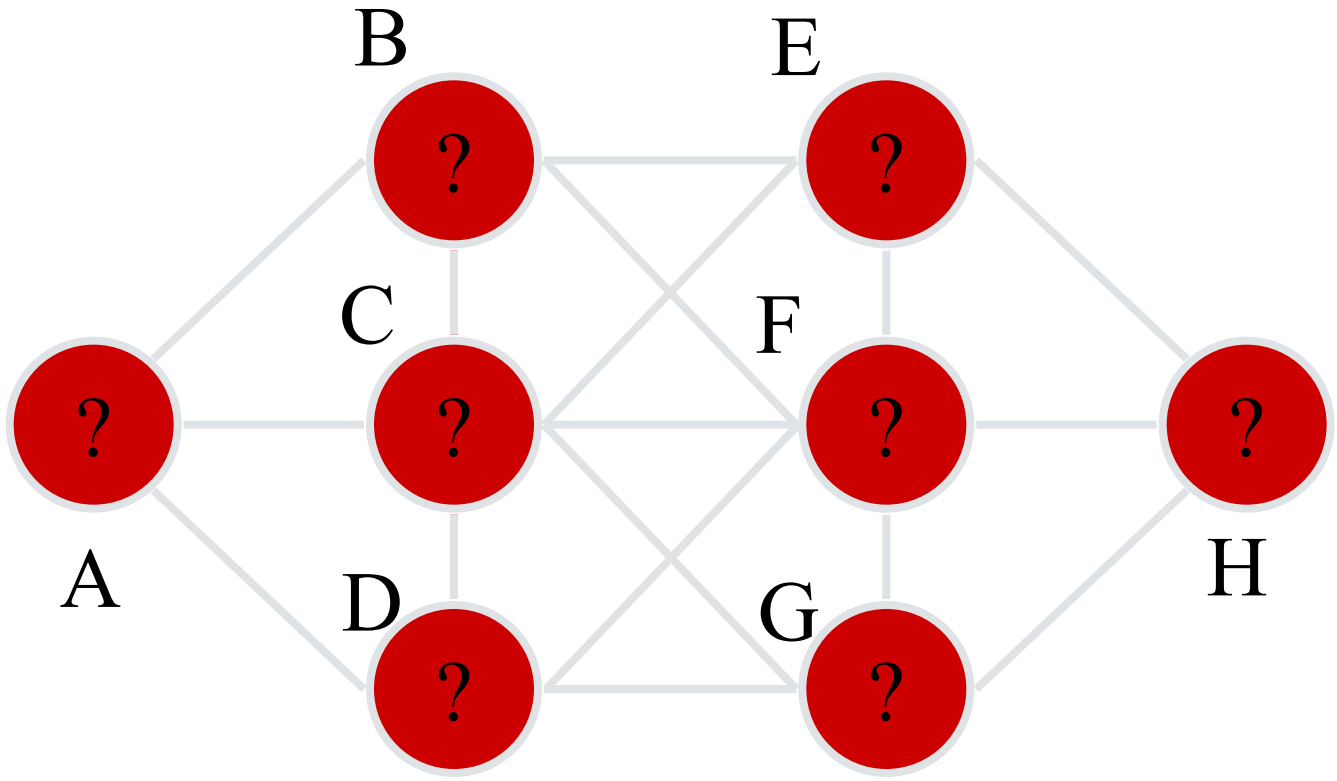
- Restart from a new random state
- Take the least worse move (increase cost by minimal amount)
- Try a new style of local search

TABU SEARCH IDEA

Local search but:

- Keep a small list of the moves we made so that we don't revisit the same state
- Keep a list of 4 pairs: the nodes that the numbers were in before we moved them

ASSIGN LABELS TO NODES



RETURN TO A STATE BEFORE WE STARTED CYCLING

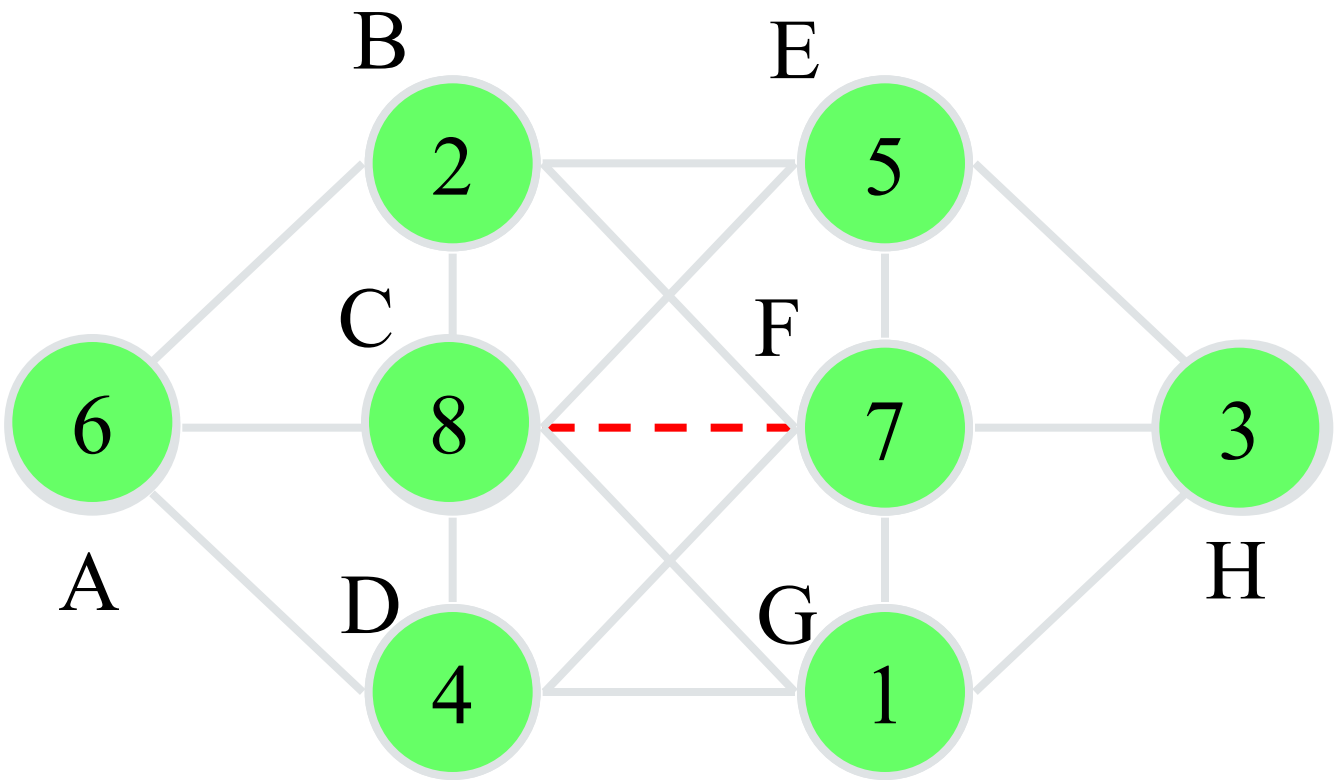
Initial State: Cost 6

Swap 1 & 7: Cost 3

Swap 3 & 8: Cost 2

Swap 6 & 7: Cost 1

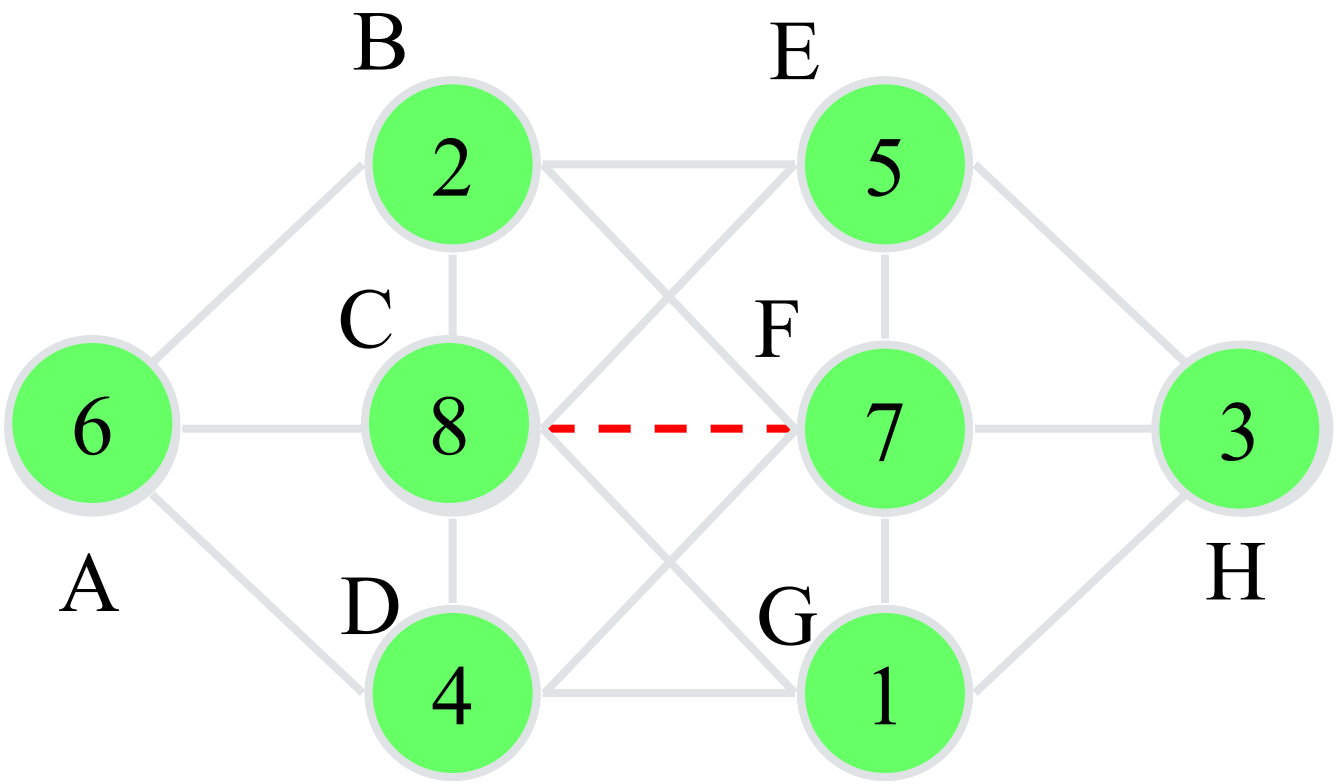
JUST SWAPPED 6 & 7: COST 1



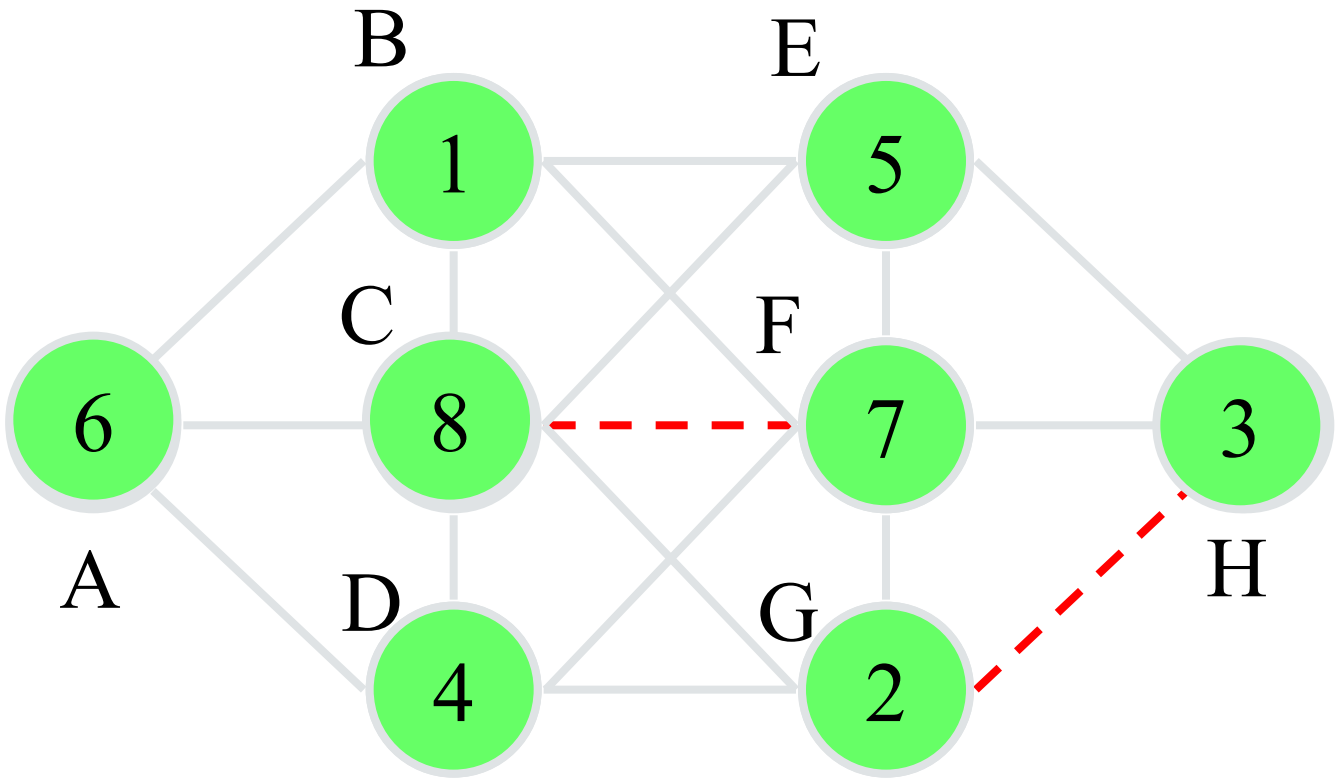
COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	1	1	1	2	2	1	1
2		0	1	2	2	1	3	1
3			0	1	1	4	1	2
4				0	2	1	3	1
5					0	2	1	2
6						0	1	1
7							0	1
8								0

JUST SWAPPED 6 & 7: COST 1



SWAP 1 & 2: COST 2



◎Tabu: [(1G,2B)]

COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	-1	0	2	1	2	2	1
2		0	1	0	2	1	1	0
3			0	1	-1	2	0	1
4				0	2	1	3	1
5					0	1	1	2
6						0	1	1
7							0	1
8								0

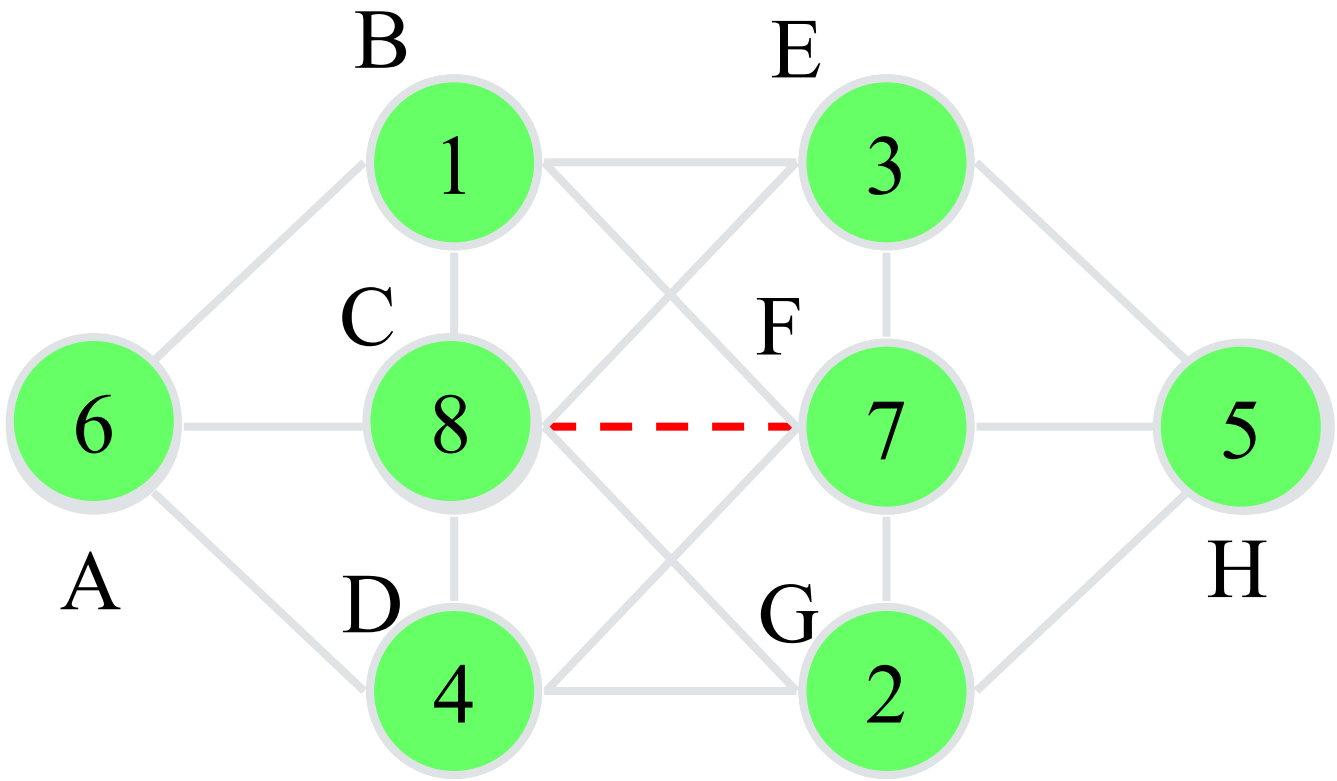
COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	X	0	2	1	2	2	1
2		0	1	0	2	1	1	0
3			0	1	-1	2	0	1
4				0	2	1	3	1
5					0	1	1	2
6						0	1	1
7							0	1
8								0

COST DIFFERENCE TABLE

	1	2	3	4	5	6	7	8
1	0	X	0	2	1	2	2	1
2		0	1	0	2	1	1	0
3			0	1	-1	2	0	1
4				0	2	1	3	1
5					0	1	1	2
6						0	1	1
7							0	1
8								0

SWAP 3 & 5: COST 1

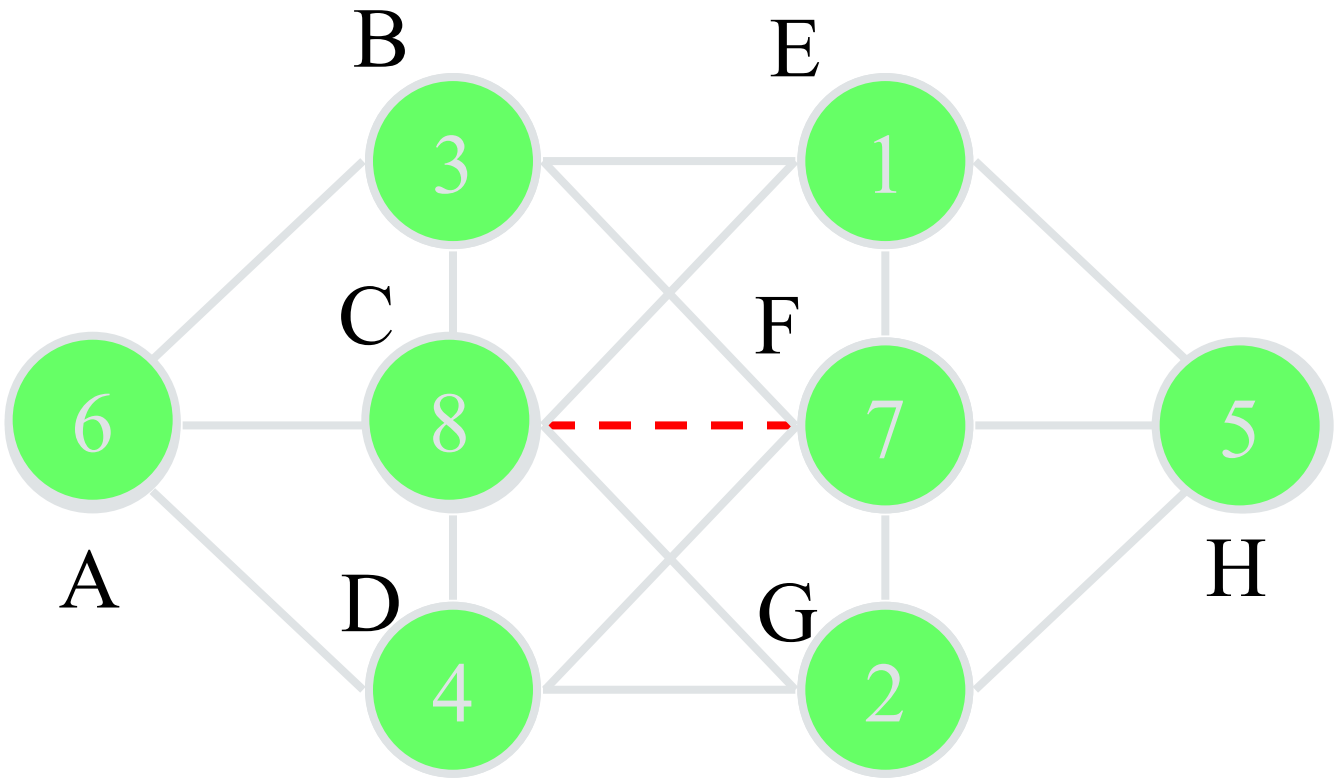


◎Tabu: [(3H,5E),(1G,2B)]

COST DIFFERENCE TABLE

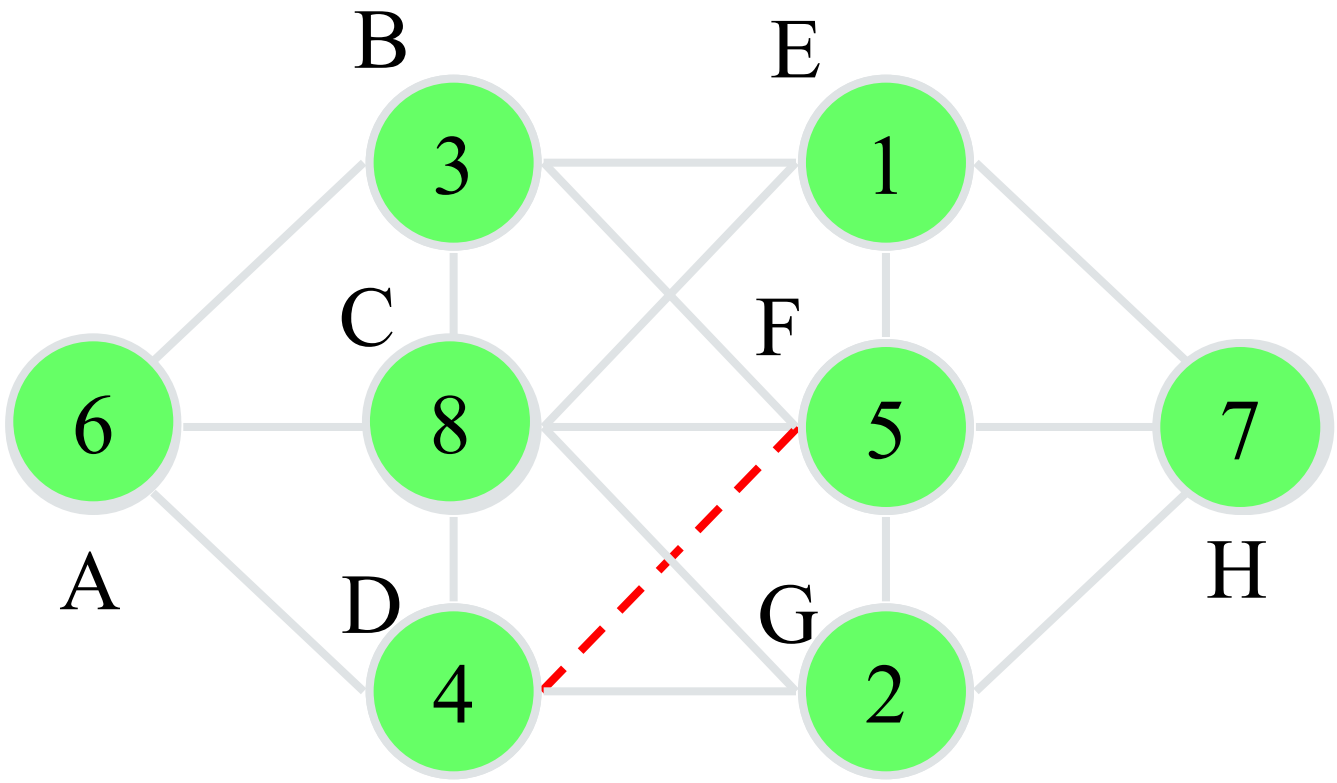
	1	2	3	4	5	6	7	8
1	0	X	0	2	2	1	2	1
2		0	2	1	2	3	2	2
3			0	2	X	3	2	2
4				0	2	1	3	1
5					0	2	0	2
6						0	1	0
7							0	1
8								0

SWAP 1 & 3: COST 1



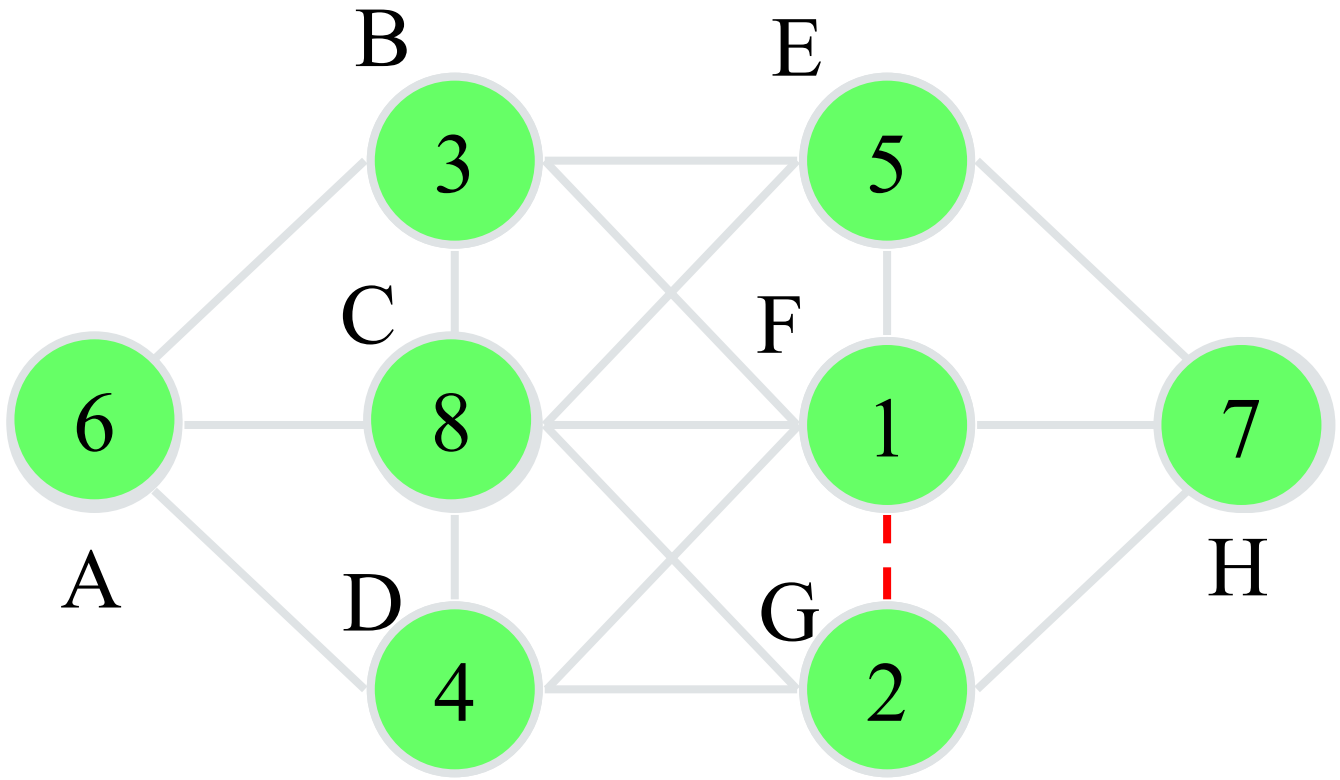
◎Tabu: [(1B,3E),(3H,5E),(1G,2B)]

SWAP 5 & 7: COST 1



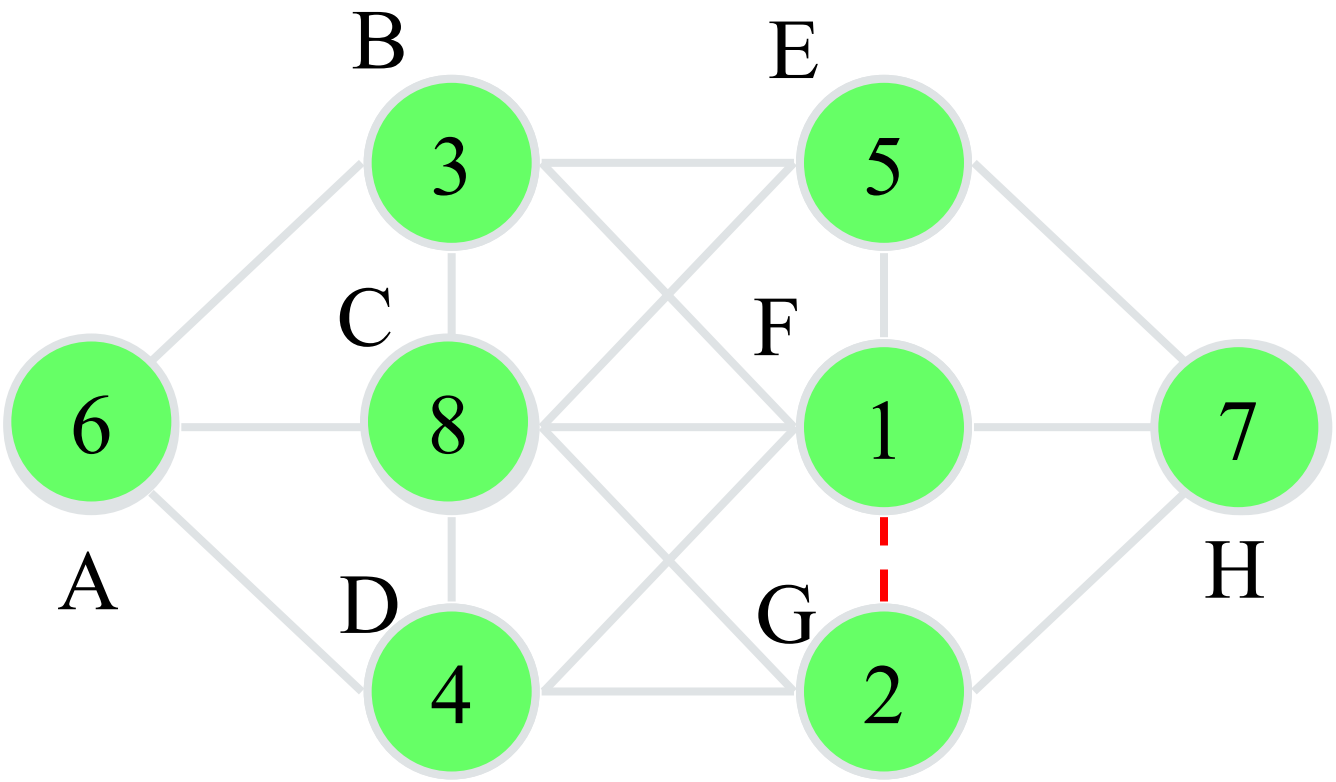
◎Tabu: [(5H,7F),(1B,3E),(3H,5E),(1G,2B)]

SWAP 1 & 5: COST 1



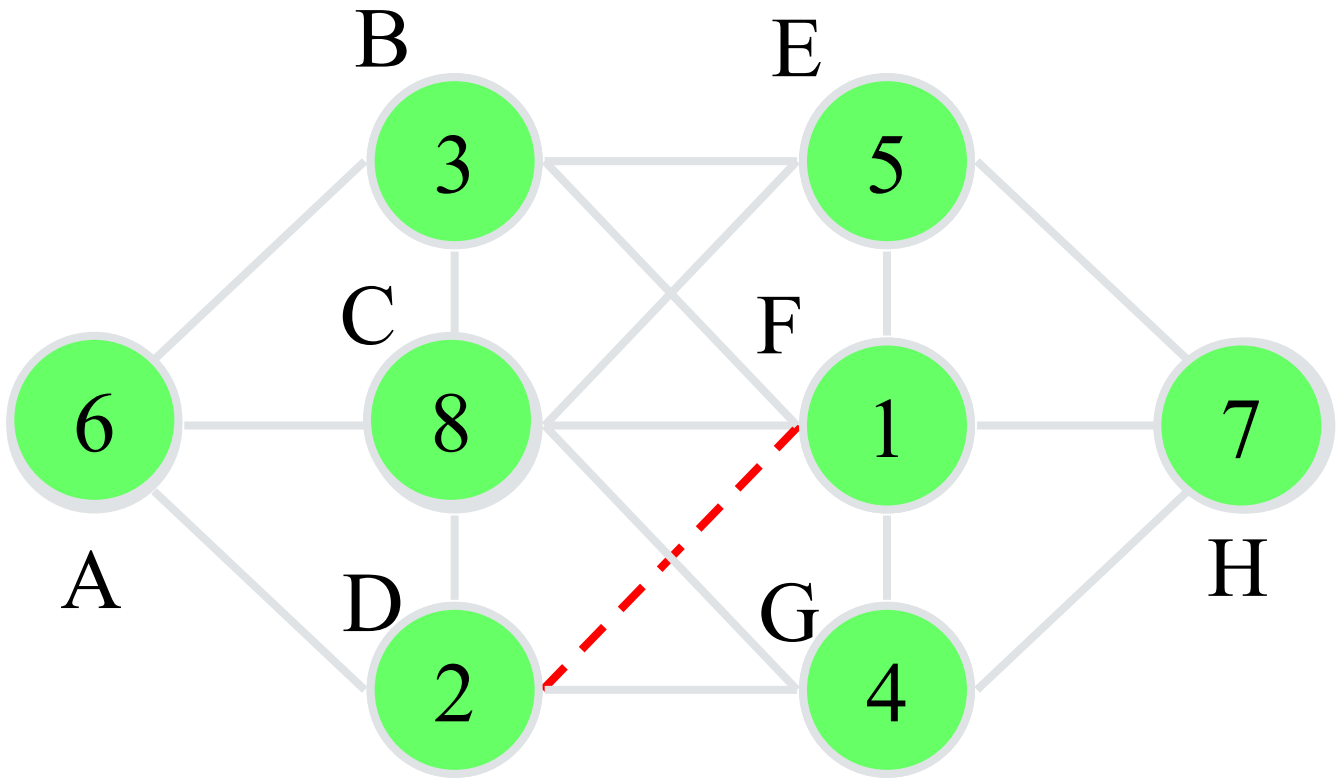
◎Tabu: [(1E,5F),(5H,7F),(1B,3E),(3H,5E),(1G,2B)]

SWAP 1 & 5: COST 1



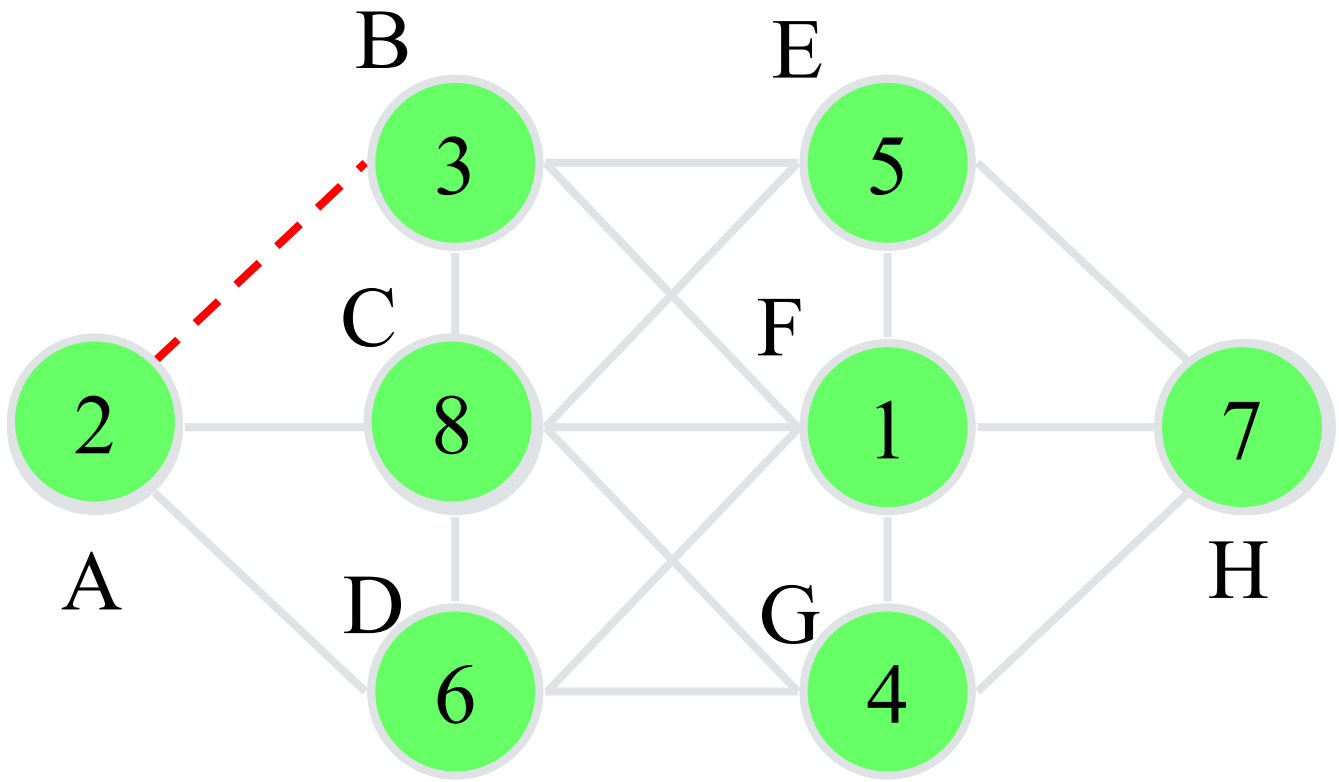
◎Tabu: [(1E,5F),(5H,7F),(1B,3E),(3H,5E)]

SWAP 2 & 4: COST 1



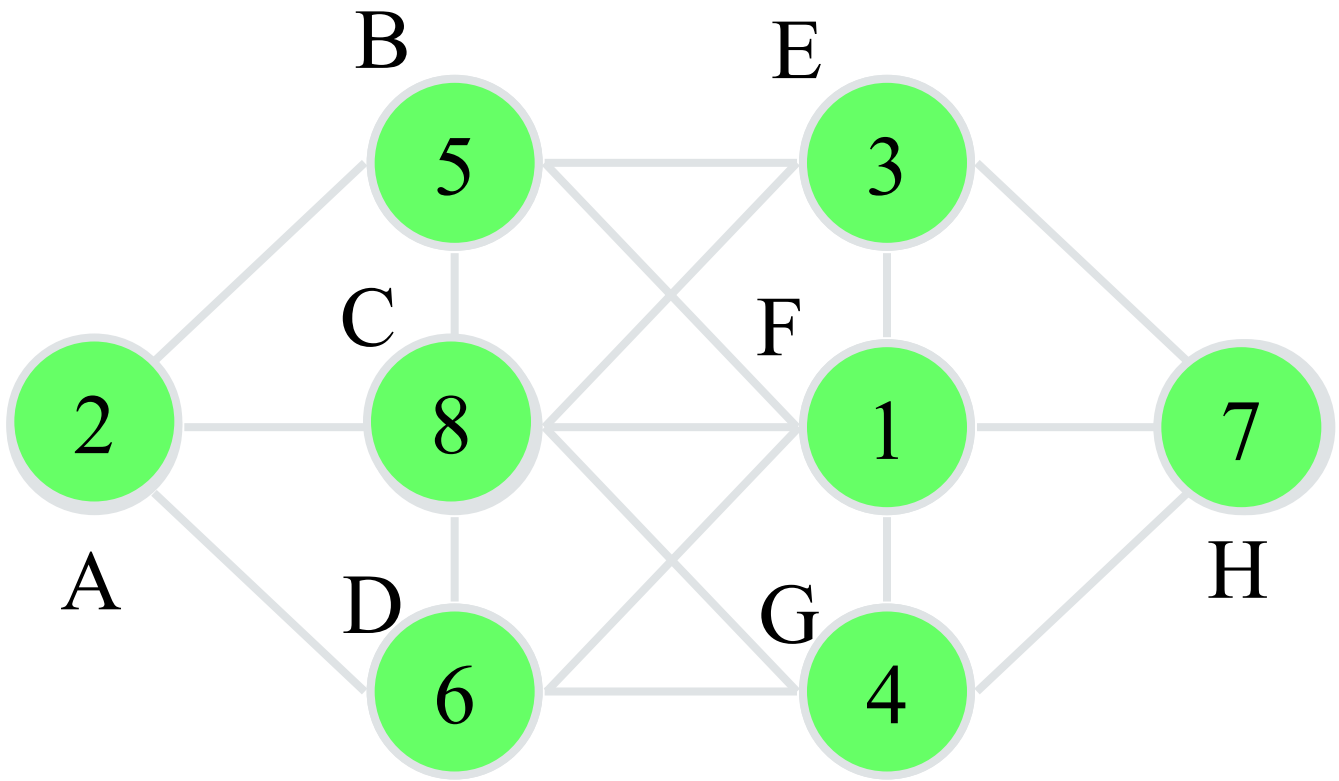
◎Tabu: [(2G,4D),(1E,5F),(5H,7F),(1B,3E)]

SWAP 2 & 6: COST 1



◎Tabu: [(2D,6A),(2G,4D),(1E,5F),(5H,7F)]

SWAP 3 & 5: COST 0



◎Tabu: [(3B,5E),(2D,6A),(2G,4D),(1E,5F)]