AI LAB 2025

ARTIFICIAL INTELLIGENCE LAB
SHAY BUSHINSKY, SPRING 2025

LAB3 PART 2: FEASIBILITY VS. OPTIMALITY

DISCRETE OPTIMIZATION PROBLEMS

GRAPH COLORING

IN THIS LECTURE

- Feasibility vs Optimality
- Graph Coloring
- Feasible and Infusible Graph Coloring
- CVRP
- Knapsack
- TTP
- ALNS
- CVRP Solution Guidelines

FEASIBILITY IN LOCAL SEARCH

- Ensures a solution meets all problem constraints, exploring the neighborhood of a current solution for better alternatives.

- Search moves between feasible solutions within the neighborhood
- Making small modifications to optimize outcomes, such as avoiding task overlaps in a scheduling problem.

ALLOWING INFEASIBLE SOLUTIONS

- Enables exploration of a broader range of potential solutions even in the wrong direction
- Potentially escaping local optima by temporarily accepting constraint violations, which can later be corrected to achieve an optimal feasible solution.

INFEASIBILITY IN GA

- Allow Infeasible Solutions
  - Punish via fitness
  - More freedom to crossover and mutation operators to

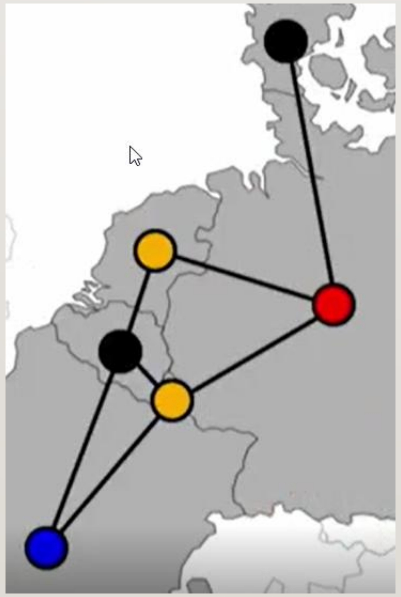
OPTIMALITY VS. FEASIBILITY

COMPLEXITY TRADEOFFS

- Neighborhood Complexity
- Objective Function Complexity
- Search Complexity

GRAPH COLORING OBJECTIVE

- Find the smallest number of colors where no adjacent countries have the same color



THE GRAPH COLORING PROBLEM

- In graph theory, a proper graph coloring is an assignment of colors to the vertices of a graph G such that no two adjacent vertices share the same color
- The graph G is defined by its vertex set V and its edge set E
- A graph coloring is a function $f: V \to C$, where C is a set of colors, and for every edge $e = (v1, v2)$ in E, $f(v1) \neq f(v2)$

A K-COLORABLE GRAPH

- If |C| = k, then G is said to be k-colorable, meaning it can be properly colored using k colors

- The **chromatic number of a graph G**, denoted as λ(G), is the minimum number of colors required to properly color the graph
- In other words, a graph G is λ(G)-colorable

GRAPH COLORING APPLICATIONS

- **Scheduling and Timetabling**
  - Vertices represent tasks, Edges represent conflicts, and colors represent time slots or resources
- **Register Allocation**
  - Vertices represent variables, Edges represent conflicts between variables, and colors represent registers
- **Frequency Assignment**
  - Vertices represent transmitters, Edges represent potential interference between transmitters, and colors represent frequencies

GRAPH COLORING APPLICATIONS

- **Map Coloring**
  - Vertices represent regions, Edges represent shared borders, and colors represent different map colors
- **Puzzle Solving (Sudoku)**
  - Vertices represent grid cells, Edges represent constraints, and Colors represent numbers
- **Social Network Analysis**
  - Vertices represent individuals, Edges represent relationships between individuals, and Colors represent different communities.

GRAPH COLORING COMPLEXITY

- Determining the chromatic number of a graph is an NP-complete problem

- Determining whether a graph is k-colorable for a given value of k is also an NP-complete problem

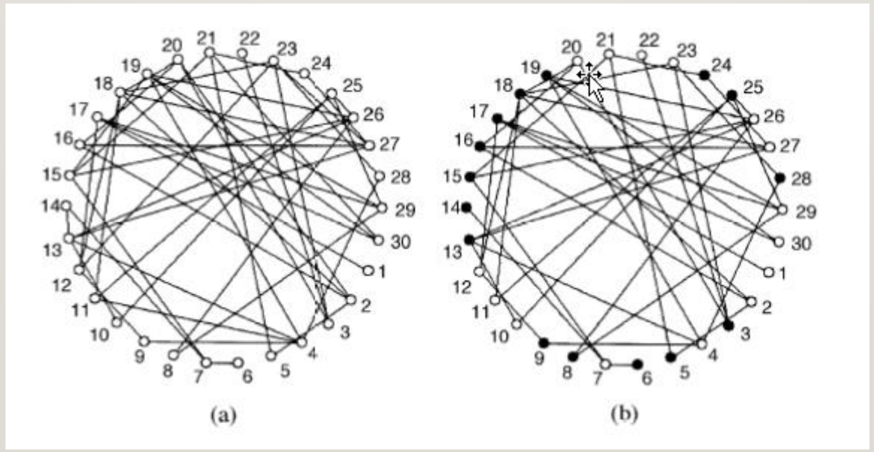
BIPARTITE GRAPHS

- A bipartite graph is a special type of graph where its vertex set can be partitioned into two disjoint sets such that every edge connects a vertex from one set to a vertex in the other set

- If $\lambda(G) = 2$, then G is a bipartite graph, as it can be properly colored with only two colors

BP-GRAPH EXAMPLE

- Graph Density: Probability of an edge between two vertices
- (a) The LEE Graph with 30 vertices 50 edges
- (b) Solution: Bi Partite Graph 38 with edges



APPLICATIONS (CONT.)

- **Matching Problems**
  - Vertices representing one set of elements and the other set of vertices representing the other set of elements
- **Collaborative Filtering and Recommendation Systems**
  - Vertices represent users and items, and Edges represent user preferences
- **Job Allocation and Task Assignment**
  - Vertices represent tasks and workers or resources, and Edges represent feasible assignments

APPLICATIONS (CONT.)

- **Data Clustering and Classification**
  - Vertices represent data points or objects and features or clusters, and Edges represent relationships or memberships
- **Social Network Analysis**
  - Vertices represent the entities, and Edges represent relationships or affiliations
- **Transportation and Logistics**
  - Vertices represent vehicles and customers, and Edges represent feasible assignments

OBJECTIVE FUNCTION AND FEASIBILITY TRADEOFFS

- **Optimization** – objective function – reducing the number of colors

- **Feasibility** – two adjacent vertices **must be** colored differently

THREE APPROACHES IN LOCAL SEARCH

1. Convert to a sequence of feasibility problems **- stay in the feasible space**
2. Allow infeasible solutions – **focus on the objective function – free to break rules**
3. **Consider both feasible and infeasible** configurations – hybrid

APPROACH 1: FOCUS ON FEASIBILITY

- Convert to a sequence of feasibility problems:

1. Find an initial solution with k colors
   - Using a Greedy algorithm
2. Remove one color, say k
   - Reassign randomly all vertices colored with k with a color in the range 1..k-1 (most likely not feasible)
3. Find a feasible solution with k-1 colors using local search(minimize violations)

- Repeat until feasibility cannot be obtained

HYBRID COLOR-KNOCKOUT WITH GA REPAIR

1. Outer Loop: Reduce Colors

- Start with a valid k-coloring (e.g. from a greedy algorithm)
- "Knock out" color k by randomly reassigning its vertices into {1…k–1}

2. Inner Loop: GA-Based Repair

- **Chromosomes:** full (k–1)-color assignments
- **Fitness:** –(number of edge-violations), so fitness = 0 means conflict-free
- **Operators:** crossover + vertex-recolor mutations (±Kempe-chain)
- **Termination:** stop as soon as any individual reaches fitness = 0

3. Repeat Until No Improvement

- If GA finds a valid (k–1)-coloring, set $k \leftarrow k-1$ and loop
- Otherwise, the previous k is the graph's chromatic number

APPROACH 2: FOCUS ON THE OBJECTIVE FUNCTION

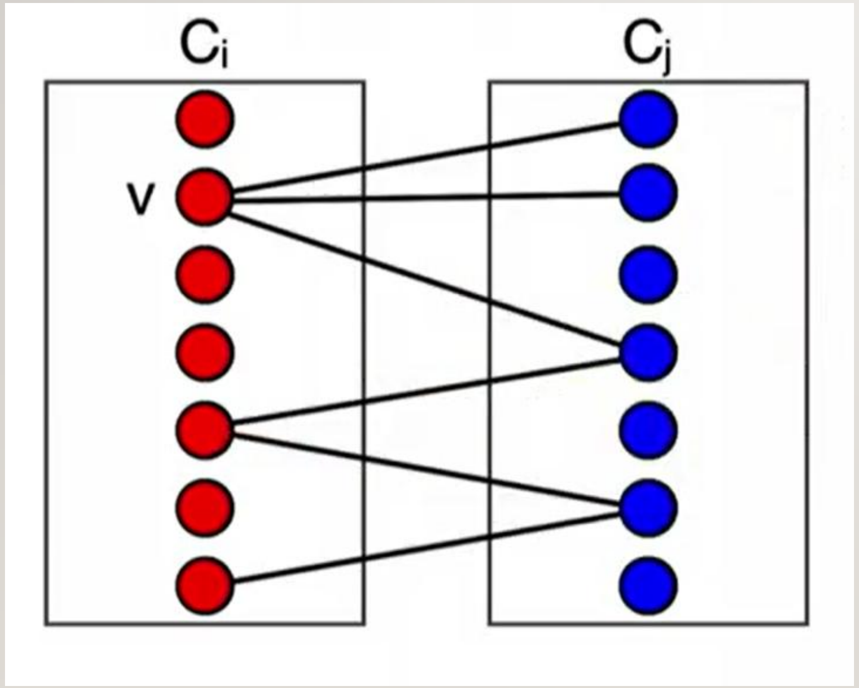
- **Always assume that a solution is feasible**
- Neighborhood
  - Change a color of a vertex
- Objective function = minimize the number of colors
- How to guide the search?
  - Changing a color of just one vertex has no impact…
  - Thus, need to introduce a new concept: ‘**Color Classes**’

COLOR CLASSES

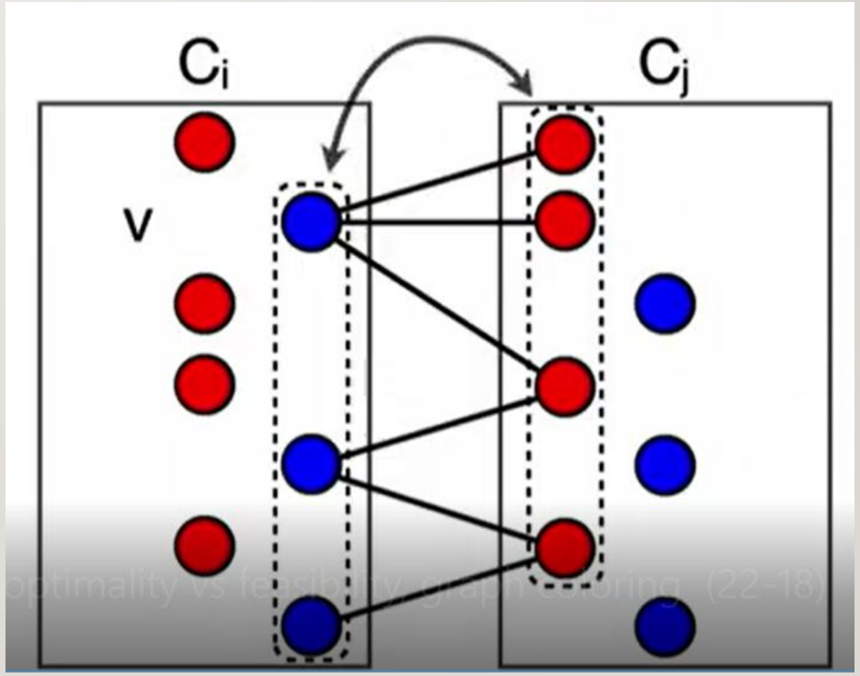
- Let Ci be the set of vertices colored with color i
- Use Ci as a proxy to guide the search:
  - Favor **large color classes**
- The new **indirect objective function**:
  - **Maximize** $\sum_{i=1}^{n} |Ci|^2$
- Yet it's difficult to maintain feasibility after moving vertices between classes…

NEIGHBORHOOD: KEMPE CHAINS

TRY TO INCREASE $|Ci|^2$ BY CHANGING V TO BLUE

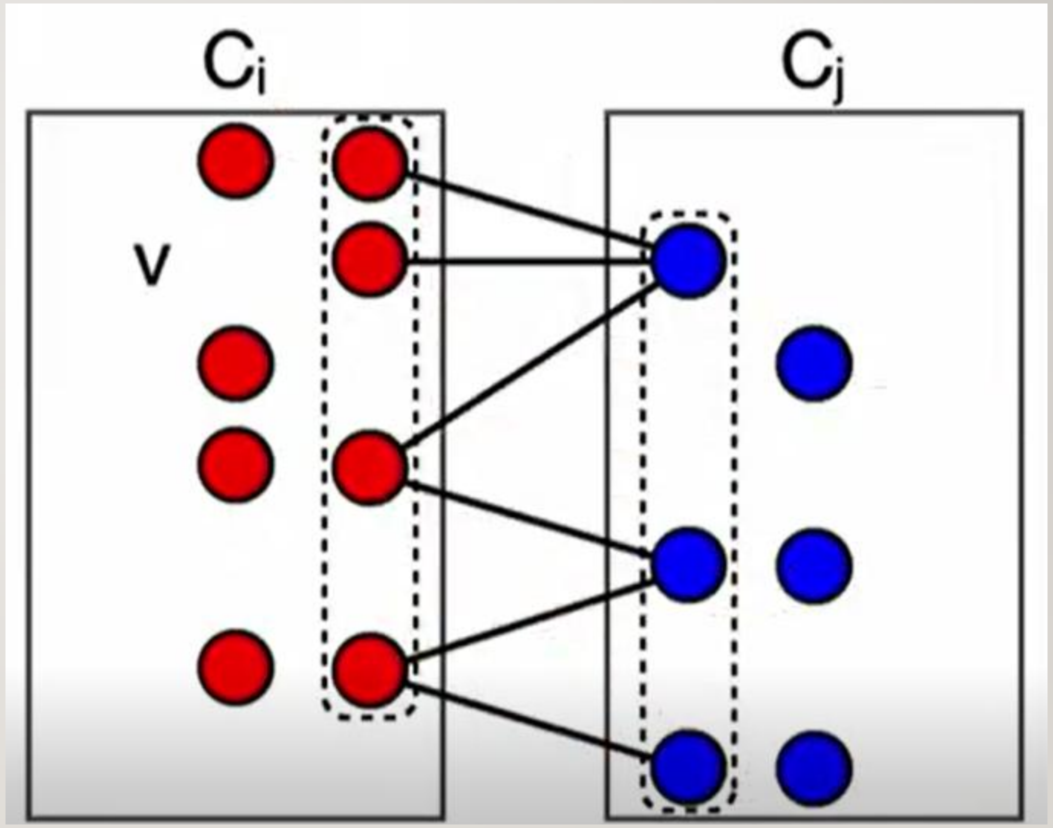


CHAIN EFFECT: SELECT A RED CHAIN INTERCHANGE IT WITH BLUE



THE INCREASED COLOR CLASSES

KEY OBSERVATION: STAYING IN THE FEASIBLE SOLUTION SPACE AFTER THE SWAP



APPROACH 3: EXPLORING BOTH FEASIBLE AND INFEASIBLE COLORING

- Focus on reducing the number of colors and on ensuring feasibility simultaneously

- Combining both objectives:
  - Make sure the local optima is feasible
  - Use a **weighted objective function** for balancing objectives:
    - Minimize $W_f f + W_o O$

INTRODUCING “BAD EDGES”

- Simple Neighborhood
  - Change the color of a vertex

- Induces **Bad Edges**:
  - A “bad edge” is one who’s connected adjacent vertices have the same color (**a violation**)
  - Let Bi be a set of bad edges between vertices with color i

THE HYBRID APPROACH

Neighborhood:

Change the color of a vertex

- On the one hand strive to **maximize** the sum of $|Ci|^2$
- While on the other hand strive to **minimize** the sum of |Bi|

COMBINED OBJECTIVE FUNCTION

- Minimize the bad edges while maximizing the color classes:
- **Minimize** $\sum_{i=1}^{n} 2|Bi||Ci| - \sum_{i=1}^{n} |Ci|^2$
- This objective function guaranties that the local minima will be a legal coloring ("feasible") by design
- Left term – feasibility constraint
- Right term - objective

THE FEASIBILITY OF LOCAL MINIMA IS GUARANTIED: A PROOF BY CONTRADICTION

- **Assume solution is an infeasible local minima**
- Consider a coloring $C_1,\ldots,C_k$
- Per assumption Bi is not empty (there is a bad edge somewhere with color i…)
- We will show that such a coloring **cannot** be a local minima:
- Consider an additional color: k+1
- Select an edge in Bi and color one of its vertices with k+1 (rather than i)
- As a result, one of the objective function components will decrease and thus the entire objective function can not be a local minima!
- **Feasibility term**: 2|Bi||Ci|-2(|Bi|-1)(|Ci|-1)=2|Bi|+2|Ci| - 2 ≥ 2|Ci|
- **Objective term:** increased by: $|Ci|^2$ - $((|Ci|-1)^2+1)$ = 2 $|Ci|$ -2
- Thus overall, the objective function decreases by the difference of both which is at least 2 Q.E.D.

THE ICN PARTITIONING ALGORITHM

REPLACING THE BAD EDGES CLASS WITH BAD VERTICES CLASS

THE ITERATIVE COLORING NEIGHBOURHOOD SEARCH (ICN)

THE IMPASSE CLASS NEIGHBORHOOD

- C.A. Morgenstern – Genetic graph coloring
- Improve a partial k coloring to a complete coloring of the same value where not all vertices are colored
- A solution S is a partition of V in k+1 color classes V1, … ,Vk;Vk+1 s.t. all classes, but possibly the last one, are **Stable Sets**

ICN PARTITIONING

- **K stable Sets** means that the first k classes constitute a **partial feasible** k coloring, while all vertices that do not fit in the first k classes are in the last one

- **Objective**: Making this last class empty which yields a complete feasible k coloring

ICN ITERATION

- To move from a solution S to a new solution S0 in the neighborhood, one can choose an uncolored vertex v from Vk+1
- Assign v to a different color class, say class h, and move to class k +1 all vertices v' in class h that are adjacent to v
- This ensures that color class h remains feasible

CHOOSING V

- The uncolored vertex v is randomly chosen, while the class h is chosen by comparing different target classes by means of an evaluating function f(S)

THE PROXY FIT FUNCTION

- Rather than simply minimizing |Vk+1|, the algorithm minimizes the global degree of the uncolored vertices:
- δ represents the degree of vertex v

THIS CHOICE FORCES VERTICES HAVING A SMALL DEGREE, WHICH ARE EASIER TO COLOR, TO ENTER THE CLASS *K + 1*

$$f(S) = \sum_{v \in V_{k+1}} \delta(v)$$

AVOIDING CYCLES

- To avoid cycling, use the following Tabu Rule:
  - A vertex v cannot take the same color h it took at least one of the last T iterations
  - For this purpose, a tabu list stores the pair (v; h)

ASPIRATION CRITERIA

- As long as the pair (v; h) remains in the tabu list, vertex v cannot be assigned to color class h

- In addition, the algorithm implements an Aspiration Criterion:
    - A tabu move can’t be performed unless it improves on the best solution encountered so far

THE CAPACITATED VEHICLE ROUTING PROBLEM

CVRP

CVRP DESCRIPTION

- Given a fleet of vehicles with fixed capacity that need to supply demand in given cities spread across a two-dimensional map.
- Find an optimal route allocation for the fleet (it's not necessary to use all vehicles) so that each vehicle leaves the warehouse, visits each city once, and returns to the warehouse.
- Ensure that by the end of the visits all demands are met.

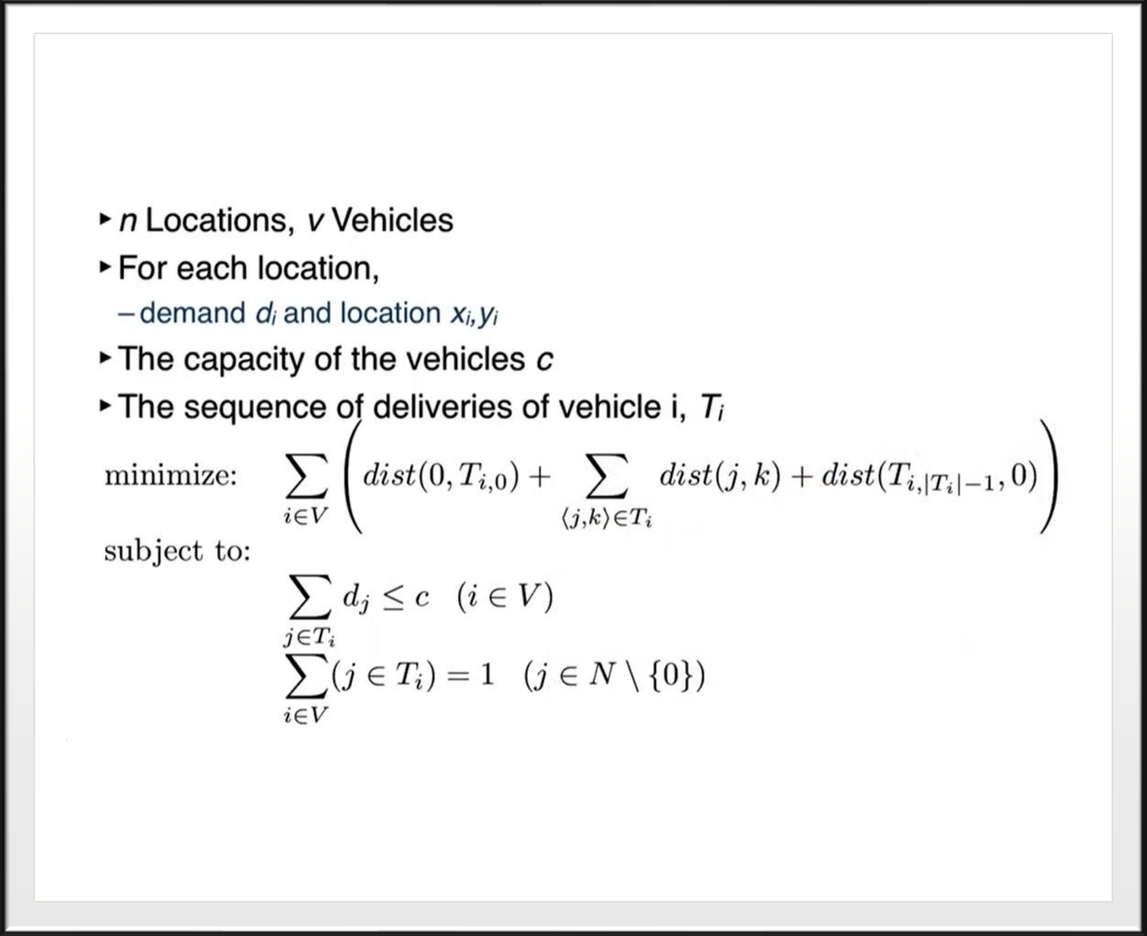
THE CVRP OPTIMAL SOLUTION

- This is a practical problem belonging to the class of NP-COMPLETE problems.

- A solution is considered optimal if it's not possible to accomplish the above with a shorter total route.

THE CVRP FORMULATION

- Choosing *v* tours from the depot that collectively visit every customer once, respect truck capacities, and keep the total driving distance as low as possible.



- *n* Locations, *v* Vehicles
- For each location,
  - demand d_i and location x_i, y_i
- The capacity of the vehicles *c*
- The sequence of deliveries of vehicle i, T_i

minimize: $$\sum_{i \in V} \left(dist(0, T_{i,0}) + \sum_{\langle j,k \rangle \in T_i} dist(j,k) + dist(T_{i,|T_i|-1}, 0) \right)$$

subject to:

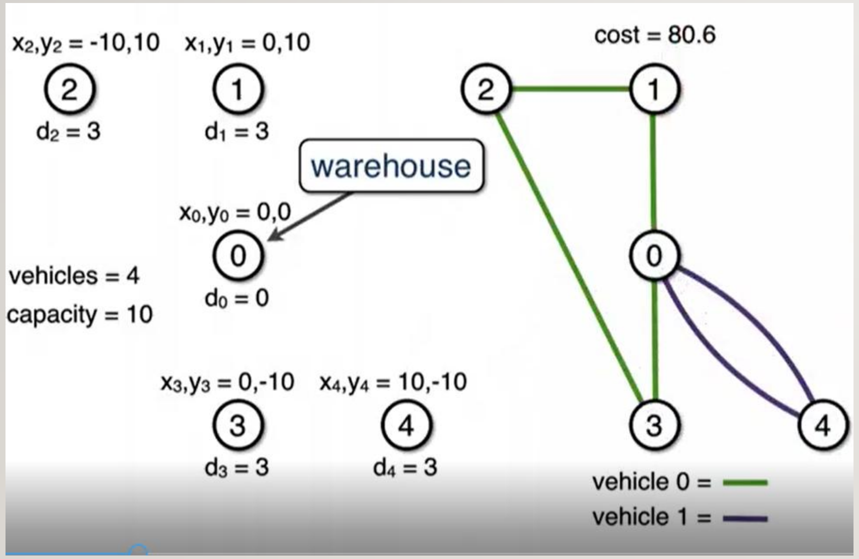
$$\sum_{j \in T_i} d_j \leq c \quad (i \in V)$$

$$\sum_{i \in V} (j \in T_i) = 1 \quad (j \in N \setminus \{0\})$$

CVRP EXAMPLE

- Given the warehouse coordinates (0,0),
- The cities' coordinates marked as (x,y)
- The demand in each city as d
- Find a route (sub-optimal) for 4 vehicles when each vehicle's capacity is 10, using only 2 vehicles:

CVRP SOLUTION



- (a) The first vehicle leaves the warehouse, visits cities 1, 2, 3, and returns to the warehouse
- (b) The second vehicle leaves the warehouse, visits city 4, and returns

- Solution value: [10+10+10+sqrt(10^2+20^2)]+ 2*sqrt(10^2+10^2) = 80.6

DISCRETE OPTIMIZATION

LINEAR PROGRAMMING RELAXATION

THE GENERAL KNAPSACK PROBLEM

- Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that:
  - the total weight is less than or equal to a given limit and
  - the total value is as large as possible

THE FORMAL DEFINITION OF THE 0-1 KNAPSACK PROBLEM

- Vi is the value of item i
- Wi is the weight of item i
- W is the knapsack capacity
- Xi is the **decision variable** per item i

- Objective function:
- Constraints:

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0,1\}$$

THE GREEDY APPROACH

Optional Selection Criteria:

1. The Smallest Wi
2. The Most valuable Vi
3. Sorting by **density Vi/Wi**
4. Combinations?

Easily feasible but is any of these optimal?

Key benefit:

Obtaining a baseline

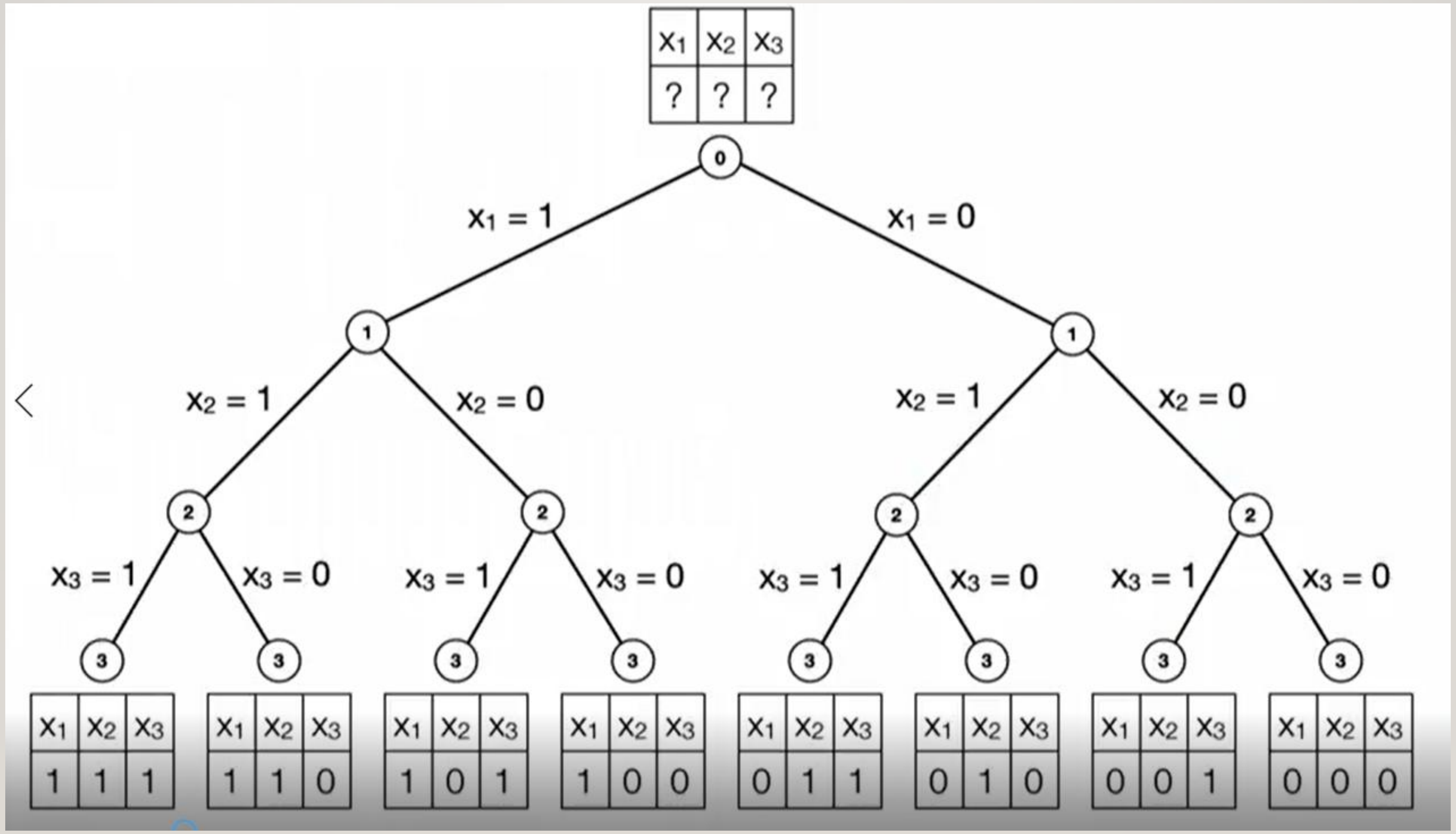
yet don’t know how far from optimal…

THE SEARCH SPACE SIZE

- A subset of all the **feasible** solutions (assignments of xi) e.g. (0,1,1,0,0,0….)

- Which is obviously exponentially large: 2^|X|

ASSIGNING THE DECISION VARIABLES EXHAUSTIVE SEARCH – DECISION TREE



BB = BRANCH AND BOUND

- Iterate between two steps –

1. **Branching** – splitting the problem into a number of sub-problems (Search Space)
2. **Bounding** – finding an **optimistic estimate** of the best solution to the sub-problem:
   1. for maximization: find a lower bound
   2. for minimization: find an upper bound

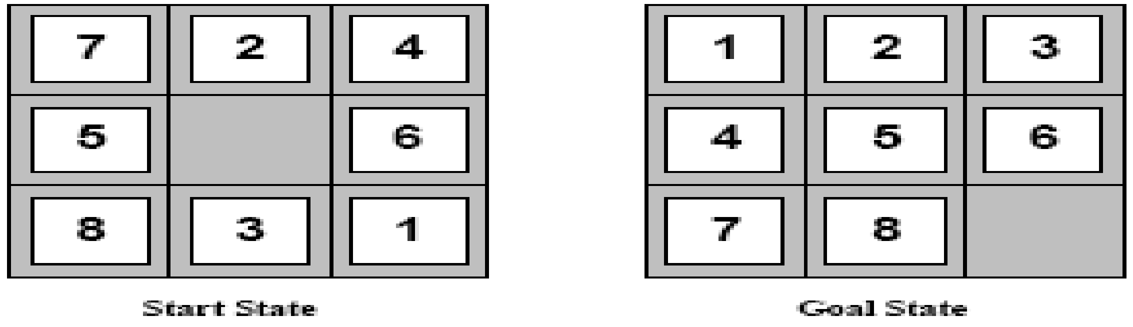
CONSTRAINT RELAXATION

- A technique to find admissible (optimistic) heuristics for solving problems using search

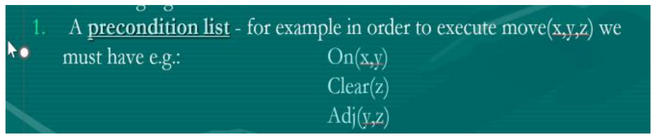
- The general principle is to derive the heuristic by **neglecting-ignoring key problem constraints**

THE N-PUZZLE RELAXATION EXAMPLES: IGNORE BLOCKING TILES

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



$h_1(S)$ =?? 6
$h_2(S)$ =?? 4+0+3+3+1+0+2+1 = 14

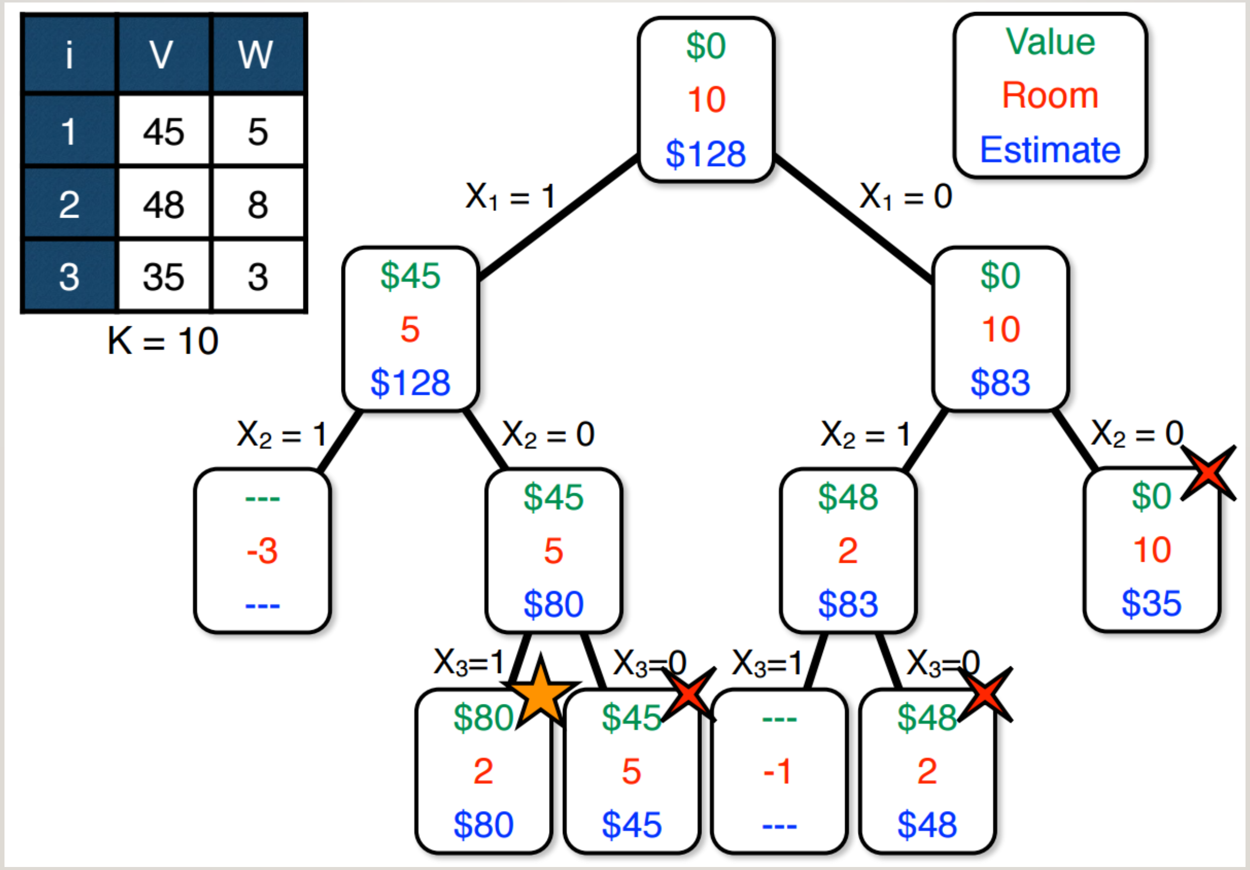

1. A precondition list - for example in order to execute move(x,y,z) we must have e.g.: On(x,y)
Clear(z)
Adj(y,z)

A KNAPSACK PROBLEM

$$\begin{array}{ll}
\text{maximize} & 45x_1 + 48x_2 + 35x_3 \\
\text{subject to} & \\
 & 5x_1 + 8x_2 + 3x_3 \leq 10 \\
 & x_i \in \{0,1\} \quad (i \in 1..3)
\end{array}$$

Integer Linear Programming – decision variables are integer, constraints are linear

1. RELAXING KNAPSACK CAPACITY



i	V	W
1	45	5
2	48	8
3	35	3

K = 10

2. RELAXING INTEGRAL ITEM SIZES

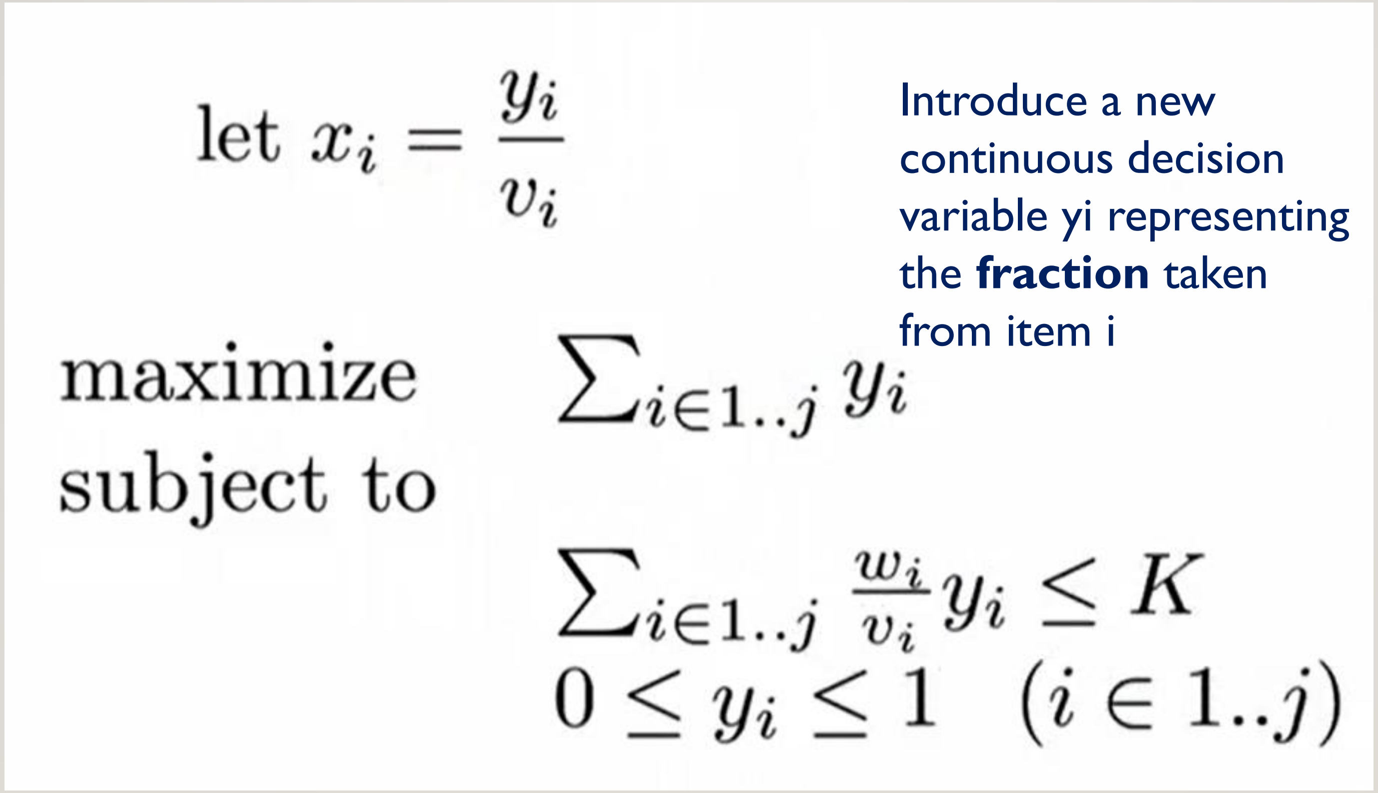
- Select the items while the capacity is not exhausted – **select a fraction of the last item** filling the knapsack

- **Normalization**: Initially order the items by decreasing value of Vi/Wi – "most value per weight" **-** a notion of "density"

OPTION 2: NEGLECTING THE INTEGRALITY CONSTRAINT

- Fractional Heuristic: **0 <= Xi <= 1**
- look at Vi/Wi:
  - V1/W1=45/5=9
  - V2/W2=48/8=6
  - V3/W3=35/3=11.7
- New order: item 3 >> item 1 >> item 2
- Greedily Select Item 3 and 1 (the most valuable)
- Complete W with ¼ of item 2
- We obtain an initial bound (estimation) of 92 - an improvement LB over 128

PROBLEM REFORMULATION

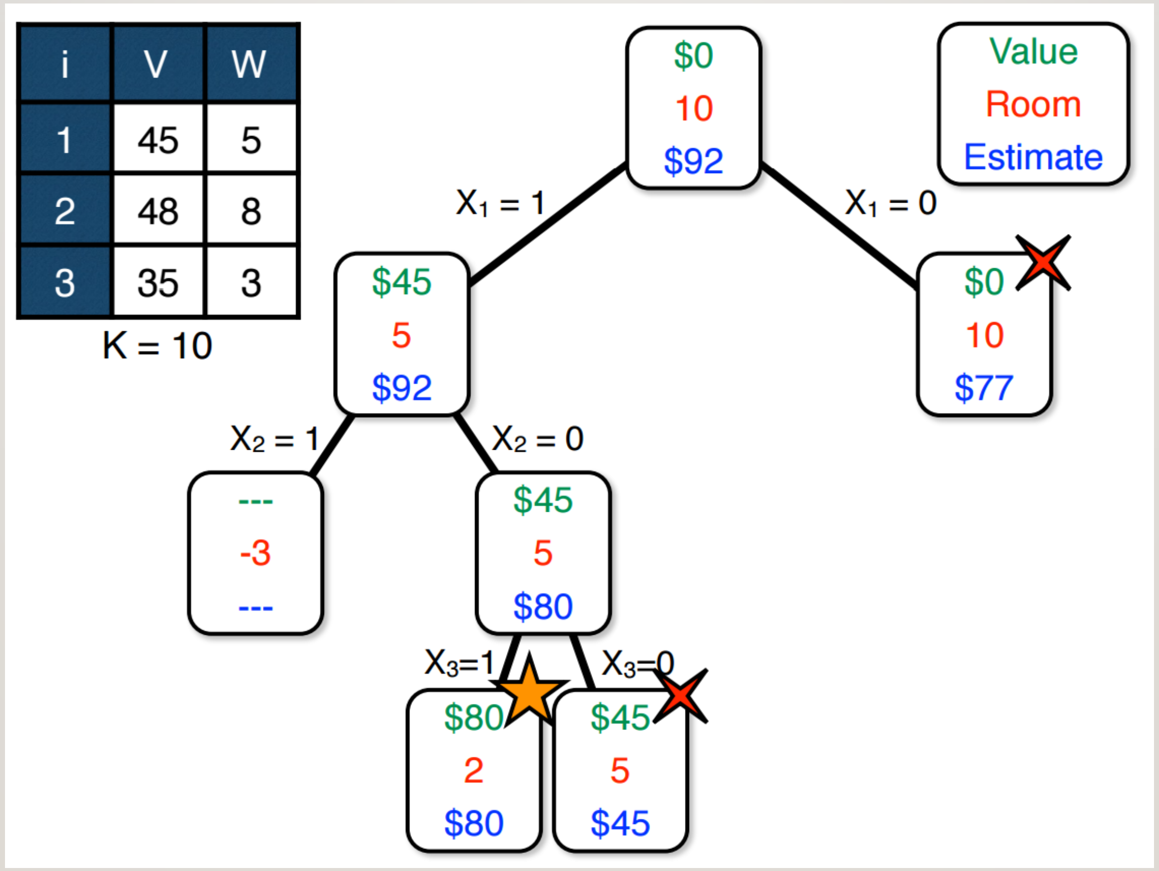


$$\text{let } x_i = \frac{y_i}{v_i}$$

Introduce a new continuous decision variable yi representing the **fraction** taken from item i

$$\begin{array}{ll} \text{maximize} & \sum_{i\in 1..j} y_i \\ \text{subject to} & \\ & \sum_{i\in 1..j} \frac{w_i}{v_i} y_i \leq K \\ & 0 \leq y_i \leq 1 \quad (i \in 1..j) \end{array}$$

RESULTS IN MORE AGGRESSIVE PRUNING



i	V	W
1	45	5
2	48	8
3	35	3

K = 10

SEARCH STRATEGIES

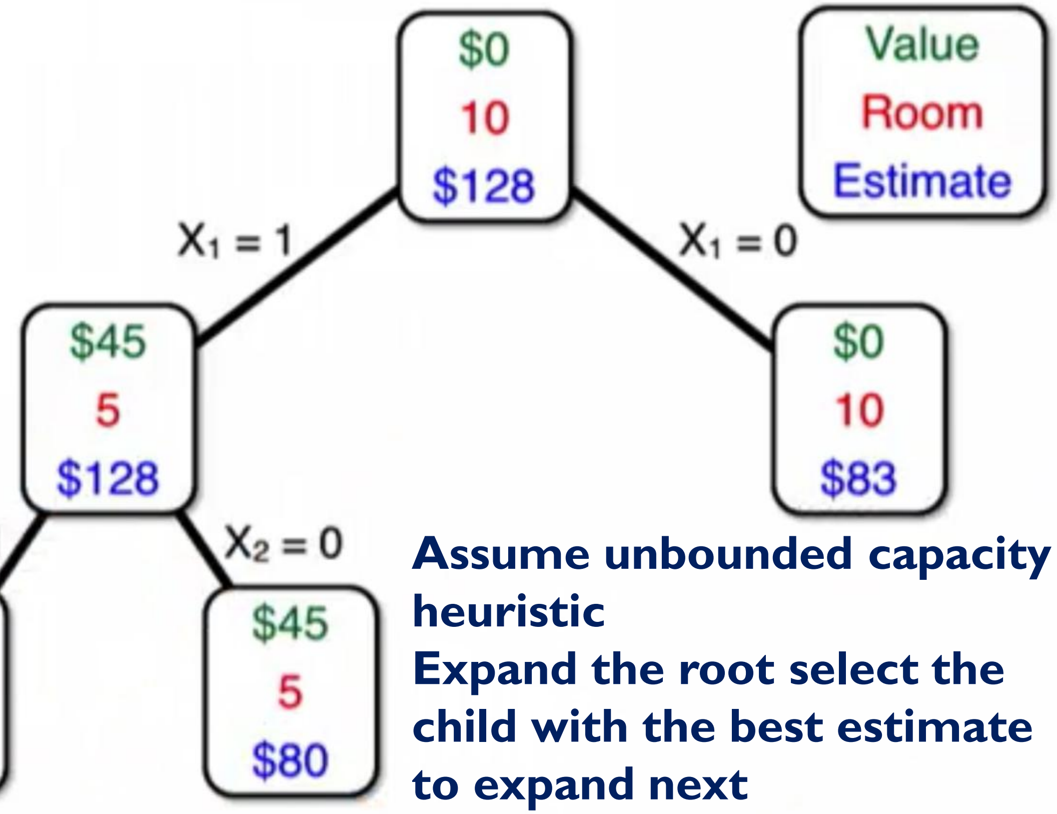
Tradeoff: good heuristic poor search performance

1. DFS BB – Memory Efficient
2. Best First Search BB – expand **only nodes better than the best evaluation so far**, stop when no node is left to expand
   - **Worst case:** Memory Inefficient
   - **Best case:** when relaxation is very good excludes most of the search tree

BEST FIRST SEARCH BB

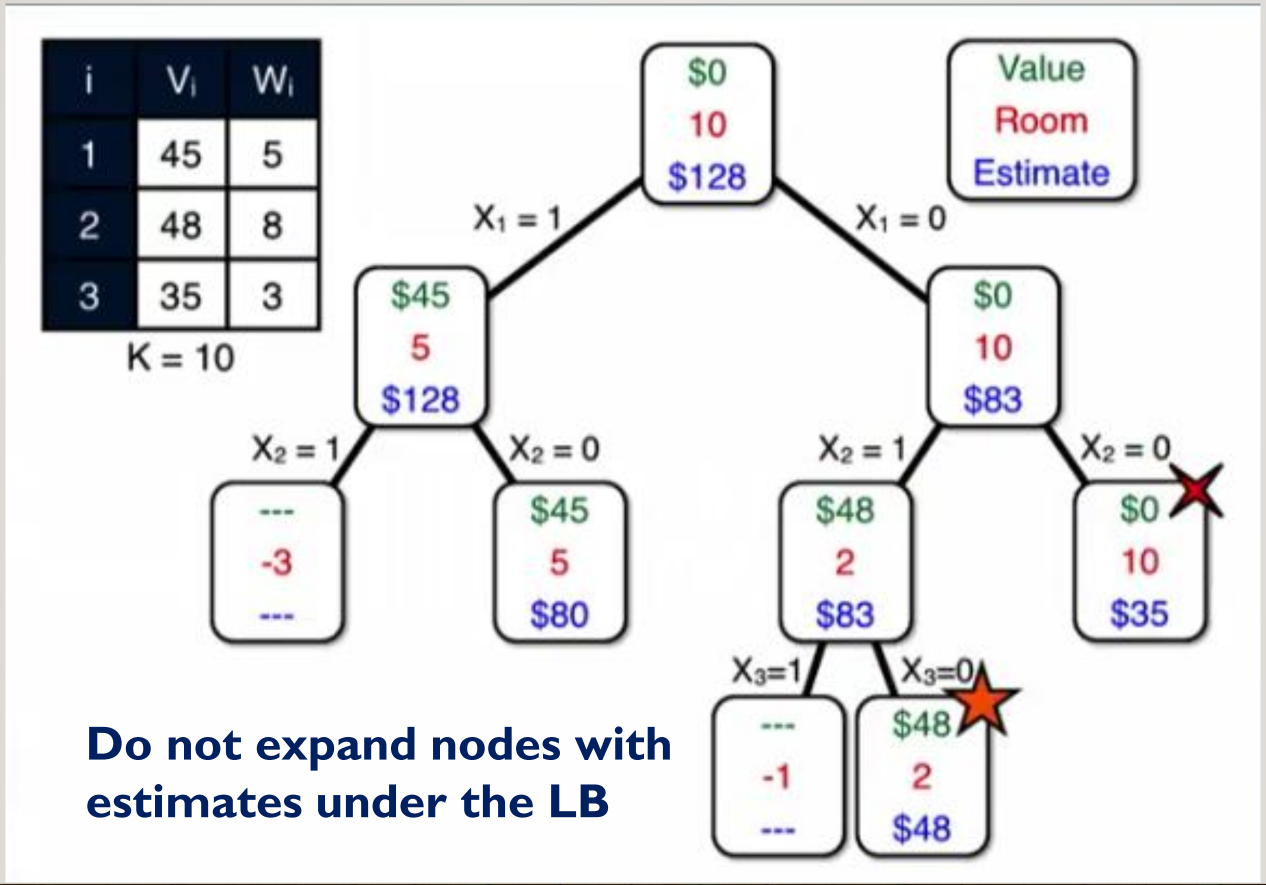
i	V_i	W_i
1	45	5
2	48	8
3	35	3

K = 10



Assume unbounded capacity heuristic
Expand the root select the child with the best estimate to expand next

BB BEST FIRST SEARCH PRUNING (2)



i	V_i	W_i
1	45	5
2	48	8
3	35	3

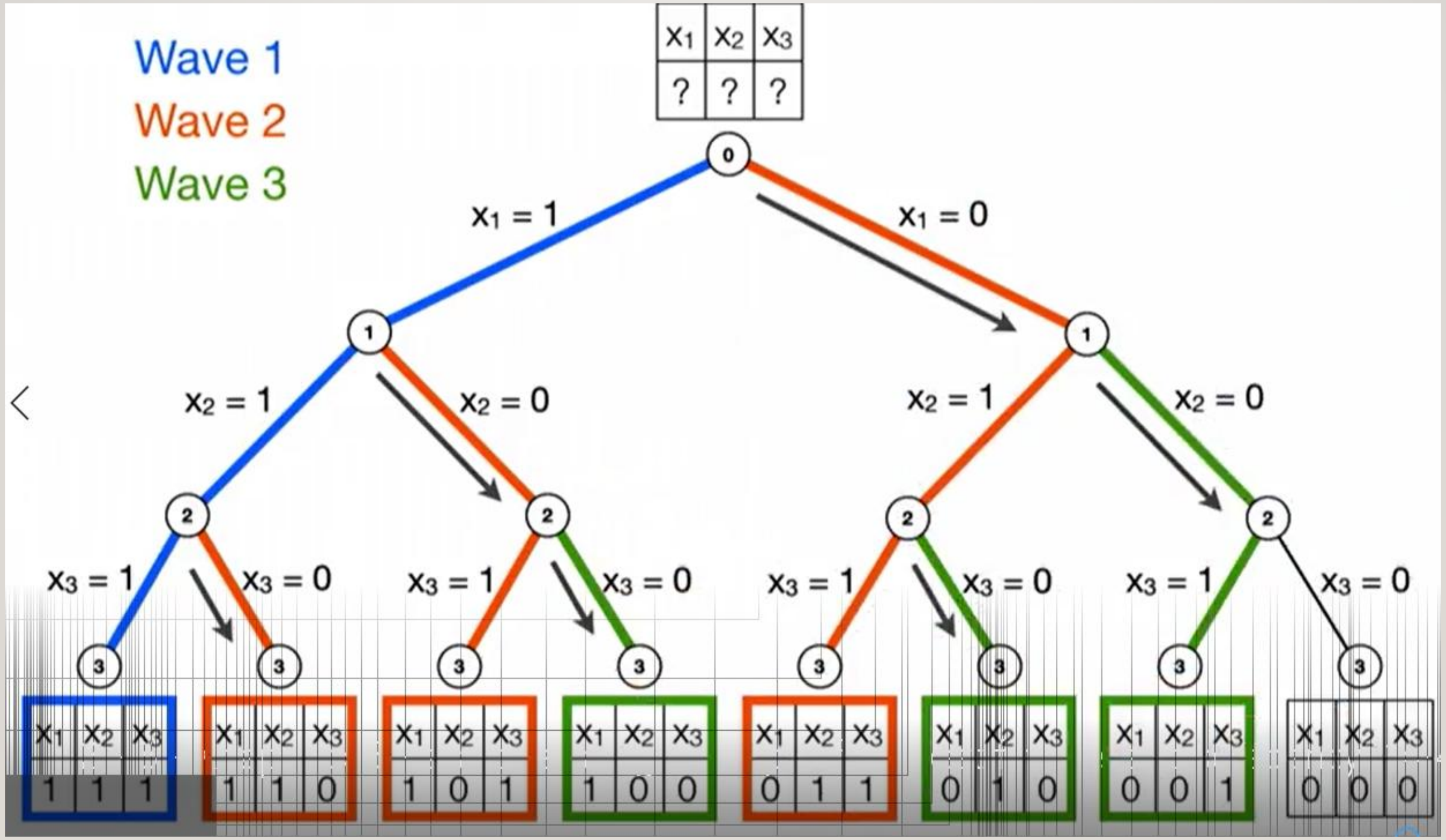
K = 10

Do not expand nodes with estimates under the LB

3. LEAST DISCREPANCY SEARCH

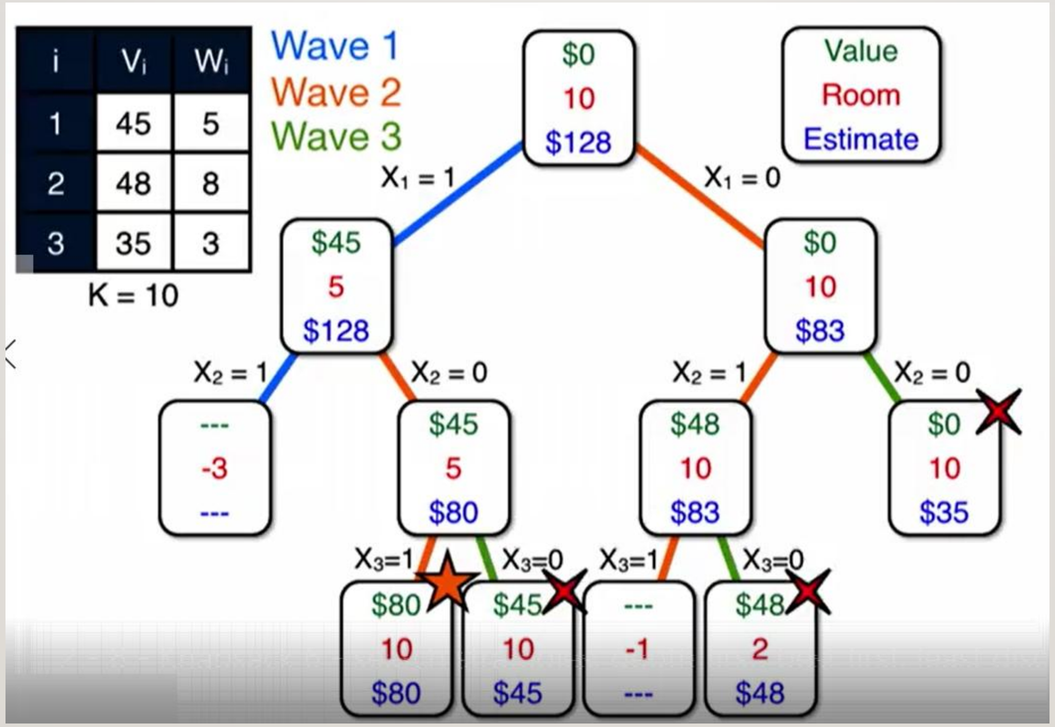
- LDS – explore the search space in waves
- Following the heuristic means branching left
- Branching right means the heuristic was mistaken –
- LDS iteration means allowing incrementally 0,1,2,… mistakes - trusting the heuristic less and less each iteration

LDS



Wave 1
Wave 2
Wave 3
x_1 x_2 x_3
? ? ?
0
$x_1 = 1$
$x_1 = 0$
1
$x_2 = 1$
$x_2 = 0$
2
$x_3 = 1$
$x_3 = 0$
3
1 1 1
1 1 0
1 0 1
1 0 0
0 1 1
0 1 0
0 0 1
0 0 0

LDS BRANCH AND BOUND



i	V_i	W_i
1	45	5
2	48	8
3	35	3

K = 10

LDS NET EFFECT

- You find your best solution (1,1,1) in **Wave 1**.
- In **Wave 2**, you only expand the one orange branch that *ties* it, and you prune all the sub-80 leaves.
- **Wave 3** is never actually entered, because no two-discrepancy branch can beat 80.
- By combining LDS's "try few wrong turns" strategy with B-and-B's "prune anyone who cannot exceed LB," you quickly zero in on the global optimum with minimal tree expansions.

DYNAMIC PROGRAMMING

- Divide and conquer algorithm
- Bottom-up computational model

- Let **o(k,j)** denote the optimal solution for a sack of capacity k

and item j is selected/examined and items thus [1..j] were selected (examined for selection already)

- We are interested in solving **o(K,n)** where K stands for the entire knapsack capacity and n means we examined all the items

FORMULATION UNDER DP

- We can now substitute the formulation to solving the **sub problem** involving up to j items of:

$$
\begin{array}{ll}
\text{maximize} & \sum_{i \in 1..j} v_i \; x_i \\
\text{subject to} & \\
 & \sum_{i \in 1..j} w_i x_i \leq k \\
 & x_i \in \{0,1\} \;\; (i \in 1..j)
\end{array}
$$

DEFINING THE ORACLE

- We assume we know the solution for **all** the subproblems o(k,j-1) and now need to consider if to add item j (o is denoted for "**oracle**") – can query them

BELLMAN'S EQUATION

- Assuming we can fit item j i.e., : Wj <= K, we thus have two cases:

1. if we decide not to add it, then the optimal solution is still o(k,j-1)
2. if we decide to add j then the optimal solution is:

   Vj + o(k-Wj, j-1)

OPTIMAL SOLUTION RECURRENCE

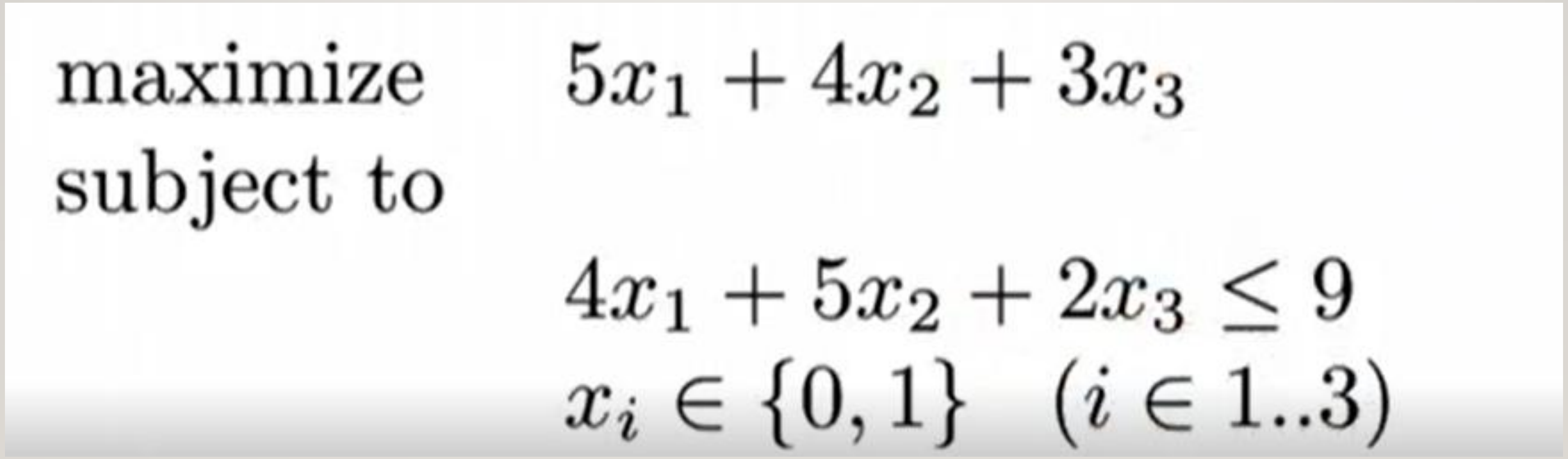
Building on the previous options that the item fits or not we can devise the following:

- $O(k,j) = \max(O(k,j-1), v_j + O(k-w_j,j-1))$ if $w_j \leq k$
- $O(k,j) = O(k,j-1)$ otherwise

Where initially we define:

- $O(k,0) = 0$ for all k

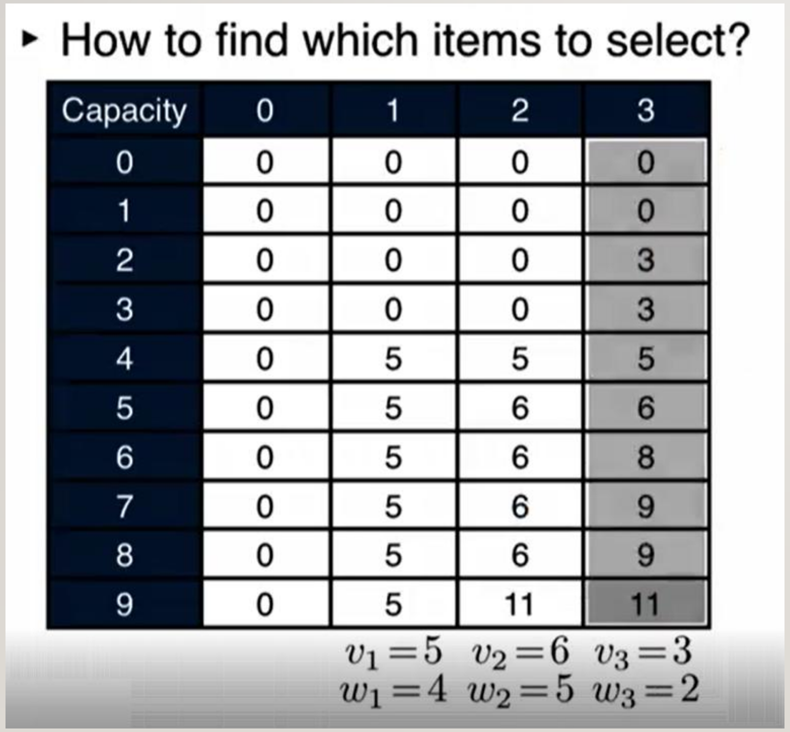
SOLVING WITH DP EXAMPLE



$$\begin{array}{ll}\text{maximize} & 5x_1 + 4x_2 + 3x_3 \\ \text{subject to} & \\ & 4x_1 + 5x_2 + 2x_3 \leq 9 \\ & x_i \in \{0,1\} \quad (i \in 1..3)\end{array}$$

DYNAMIC PROGRAMMING SOLUTION: BOTTOM-UP COMPUTATION

- Solve by adding one more item at a time:



▸ How to find which items to select?

Capacity	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	3
3	0	0	0	3
4	0	5	5	5
5	0	5	6	6
6	0	5	6	8
7	0	5	6	9
8	0	5	6	9
9	0	5	11	11

$v_1=5$ $v_2=6$ $v_3=3$
$w_1=4$ $w_2=5$ $w_3=2$

Items are ordered 1,2,3

- Col 0 = can select no items
- Col 1 can select only 1 item

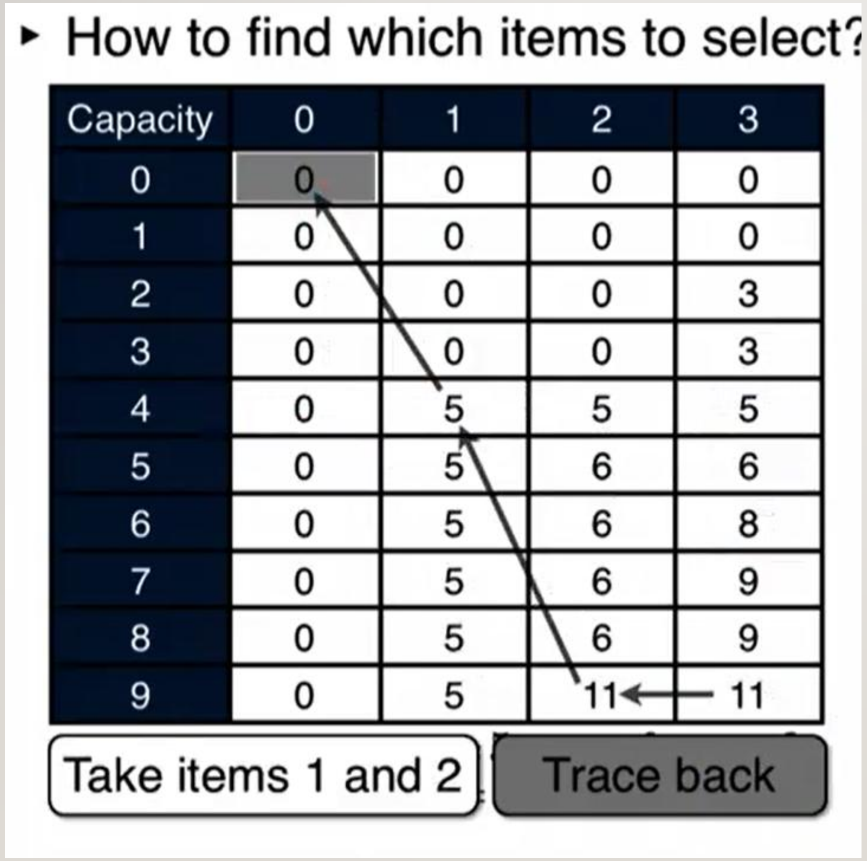
interesting at row 4 where you can select item 1

- Col 2 can select two items

until row 5 no dilemma can select only item 1 but in tow 5 have a choice between item 1,2 so you select 2 because it has the max val (according to Bellman's Eq.)

- Col 3 can select item 3 starting from 2 as w3=2 in row 4 we can select potentially item 3

EXTRACTING THE OPTIMAL SOLUTION



- How to find which items to select?

Capacity	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	3
3	0	0	0	3
4	0	5	5	5
5	0	5	6	6
6	0	5	6	8
7	0	5	6	9
8	0	5	6	9
9	0	5	11	11

Take items 1 and 2

Trace back

Retracing the solution

- From the corner optimal solution look to the left
- if value is equal means we didn't select the item (3)
- Next, we see its different meaning we selected the item (2)

And compute the remaining capacity of knapsack

- Go accordingly to item 1 sack has the capacity and again the left value is different meaning we selected item 1

ADAPTIVE LARGE NEIGHBORHOOD SEARCH ALNS

THE TOURNAMENT TOUR PROBLEM -A SCHEDULING APPLICATION

TTP

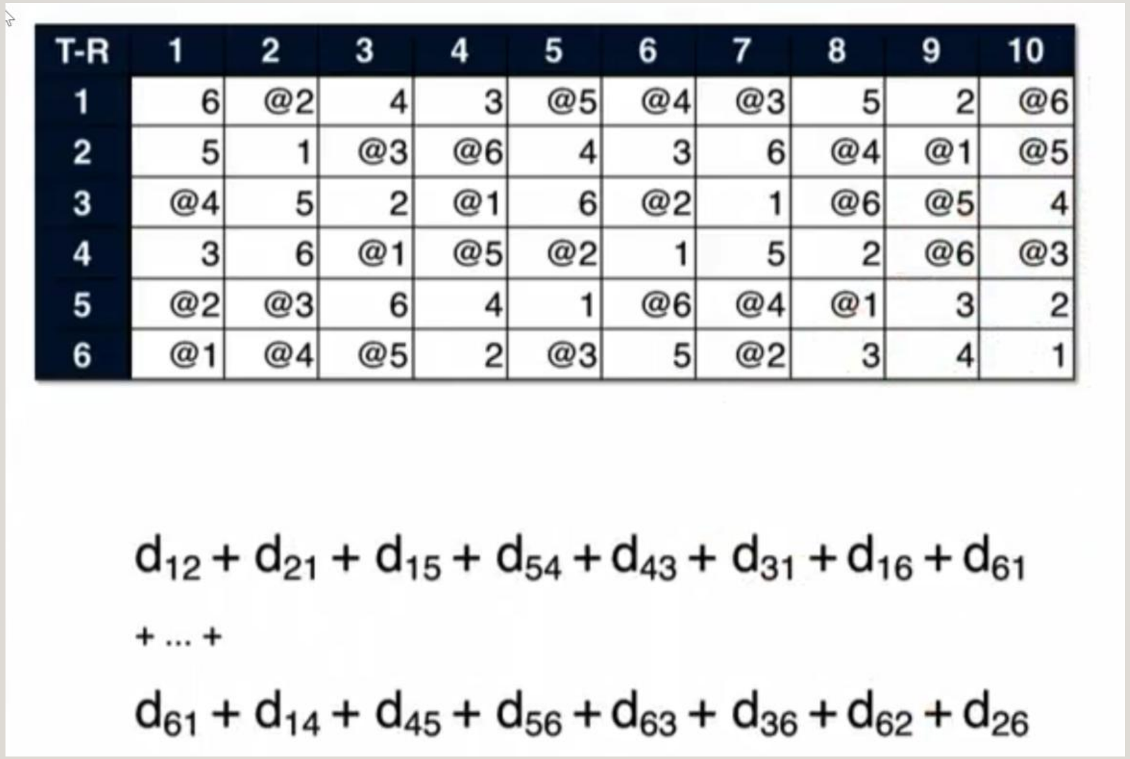
THE TRAVELING TOURNAMENT PROBLEM

- **Complex feasibility**: # games home/away
- **Objective function**: minimize travel distance
- **Input teams**
- Matrix of d distances between cities
- Season divided to weeks

TTP CONSTRAINTS

- Each team needs to play each team home and away (Double Round Robin)
- **Constraints**:
  1. Limit: 3 successive games at most at home and away
  2. No successive repeating games against same team

COMPLEX NEIGHBORHOODS TTP



T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@3	@6	4	3	6	@4	@1	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	@2	1	5	2	@6	@3
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

$$d_{12} + d_{21} + d_{15} + d_{54} + d_{43} + d_{31} + d_{16} + d_{61}$$

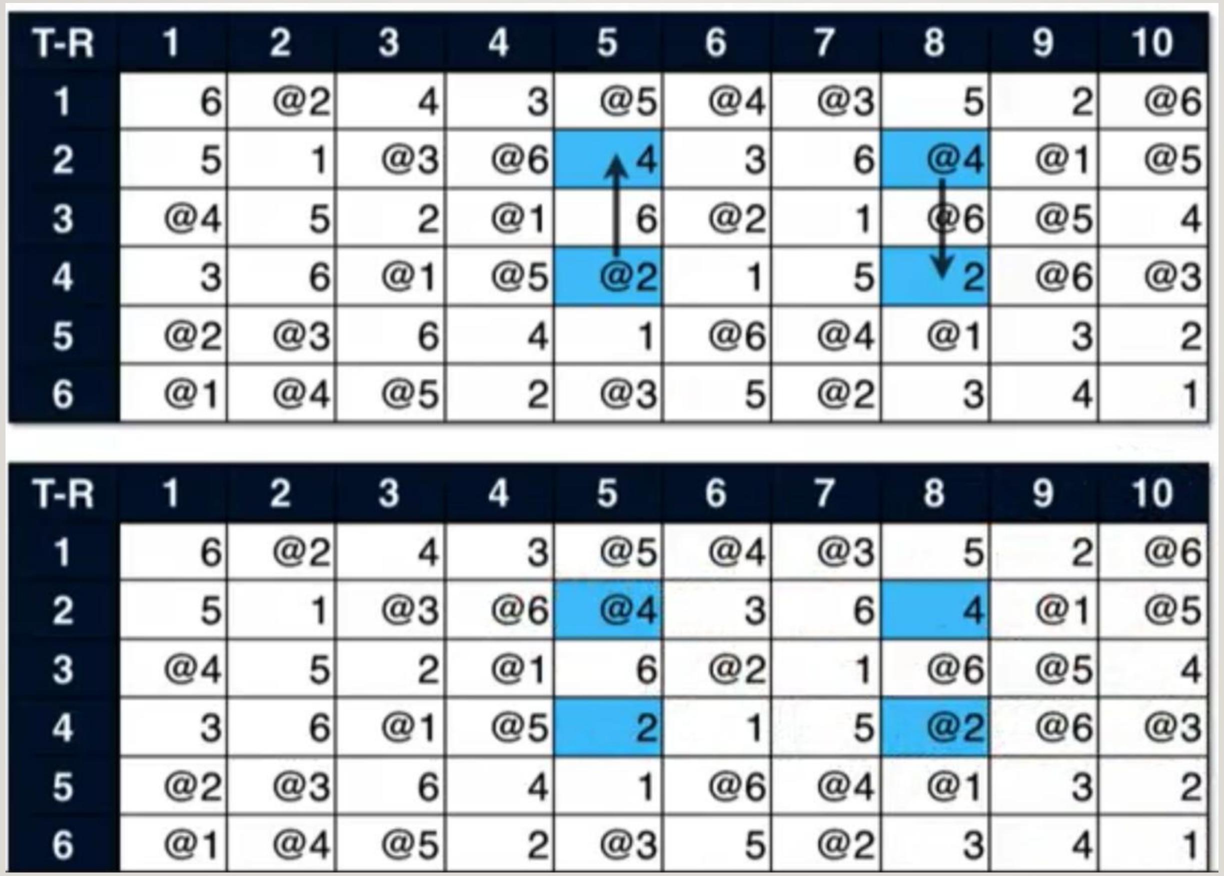
$$+ \dots +$$

$$d_{61} + d_{14} + d_{45} + d_{56} + d_{63} + d_{36} + d_{62} + d_{26}$$

TTP OPERATORS

1. Swap Homes
2. Swap Rounds
3. Swap Teams
4. Partial Swap Rounds
5. Partial Swap Teams

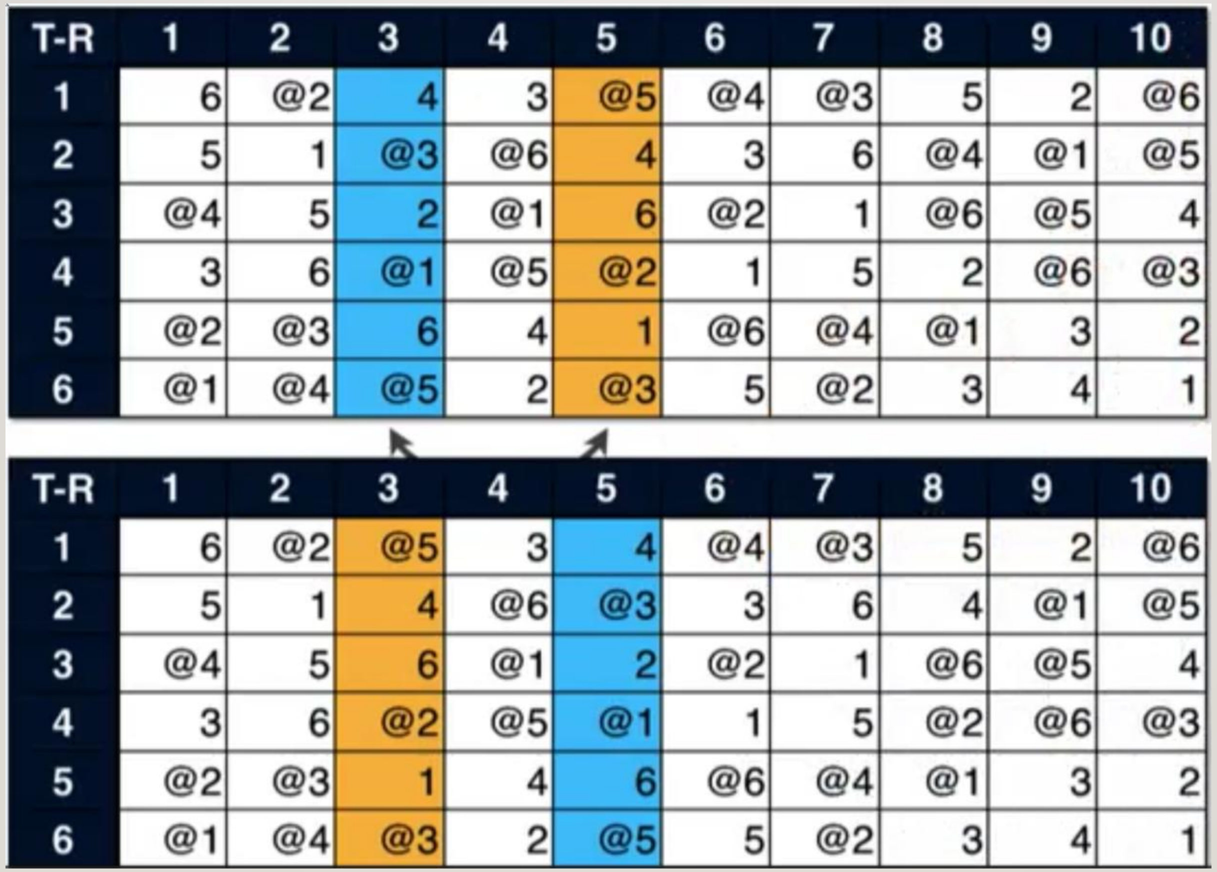
OP 1: SWAP HOMES



T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@3	@6	4	3	6	@4	@1	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	@2	1	5	2	@6	@3
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@3	@6	@4	3	6	4	@1	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	2	1	5	@2	@6	@3
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

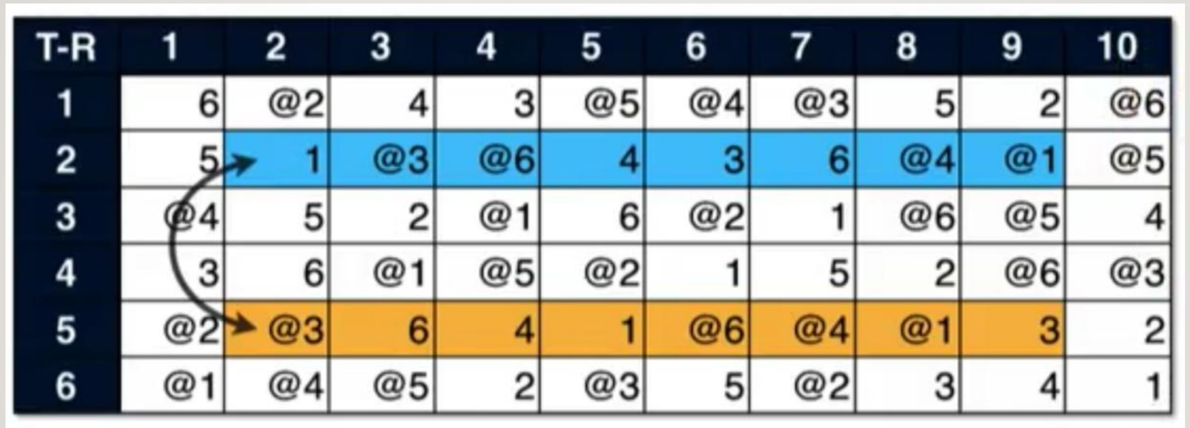
OP 2: SWAP ROUNDS



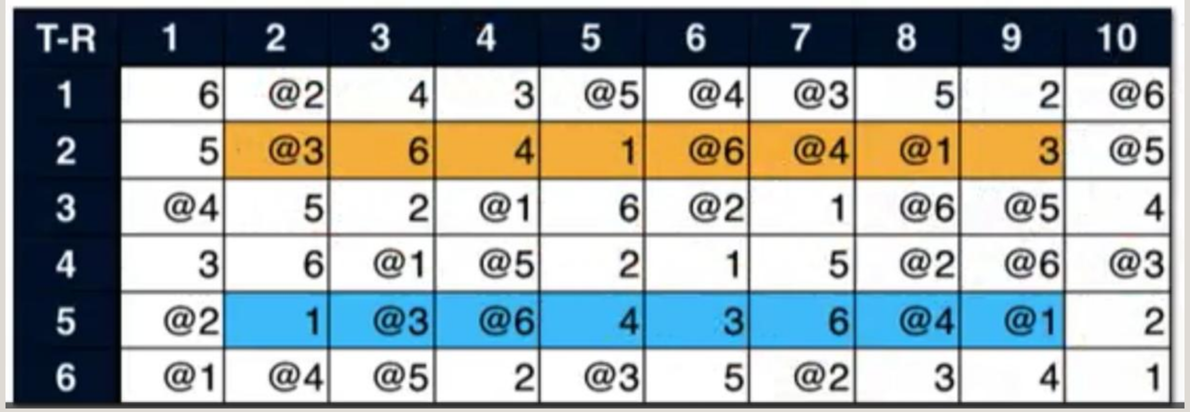
T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@3	@6	4	3	6	@4	@1	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	@2	1	5	2	@6	@3
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	@5	3	4	@4	@3	5	2	@6
2	5	1	4	@6	@3	3	6	4	@1	@5
3	@4	5	6	@1	2	@2	1	@6	@5	4
4	3	6	@2	@5	@1	1	5	@2	@6	@3
5	@2	@3	1	4	6	@6	@4	@1	3	2
6	@1	@4	@3	2	@5	5	@2	3	4	1

OP 3: SWAP TEAMS (1)

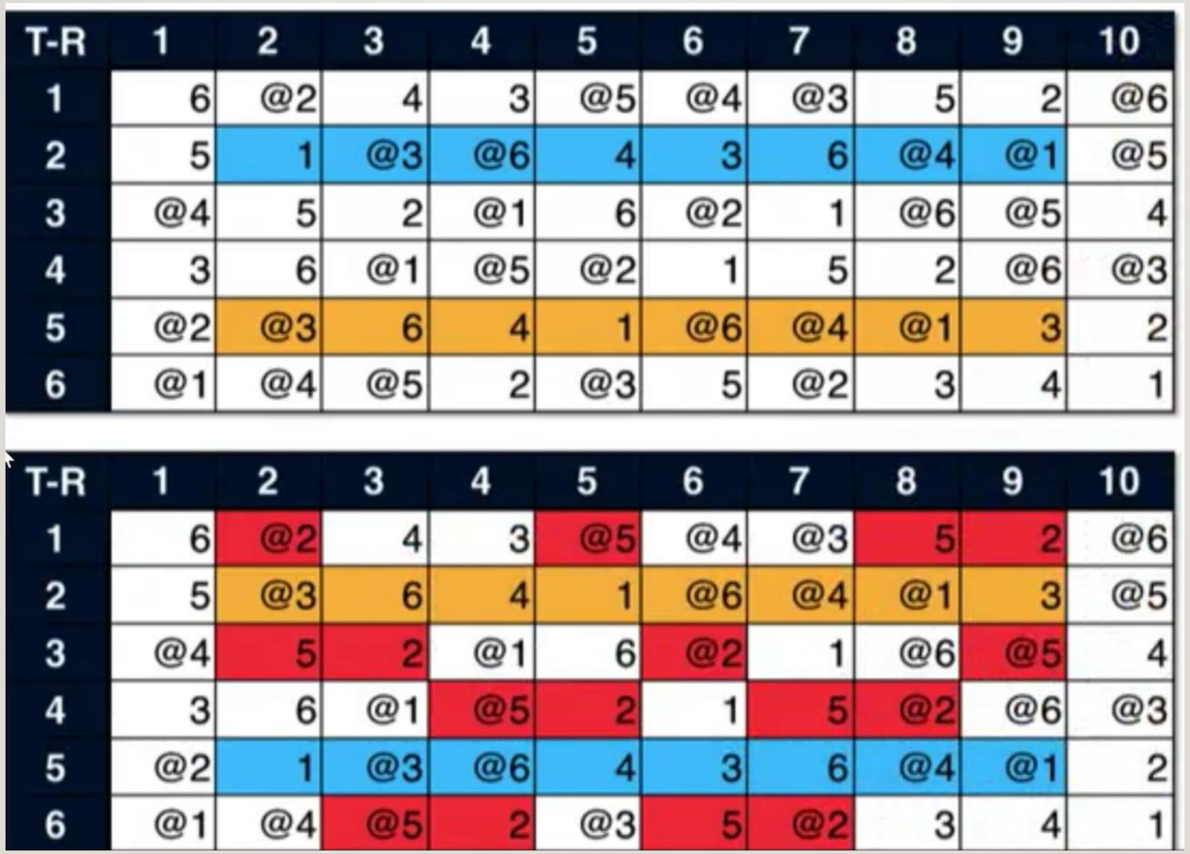


T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@3	@6	4	3	6	@4	@1	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	@2	1	5	2	@6	@3
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1



T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	@3	6	4	1	@6	@4	@1	3	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	2	1	5	@2	@6	@3
5	@2	1	@3	@6	4	3	6	@4	@1	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

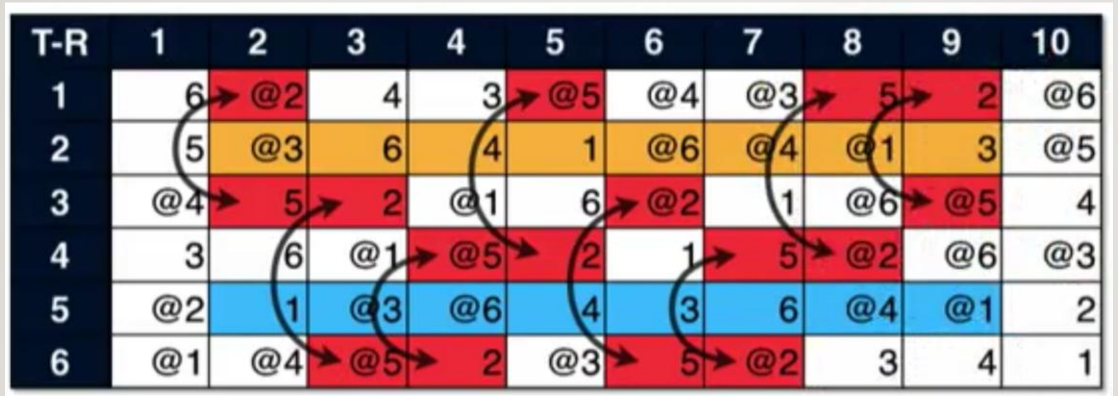
OP 3: SWAP TEAMS (2) OPP SWAP



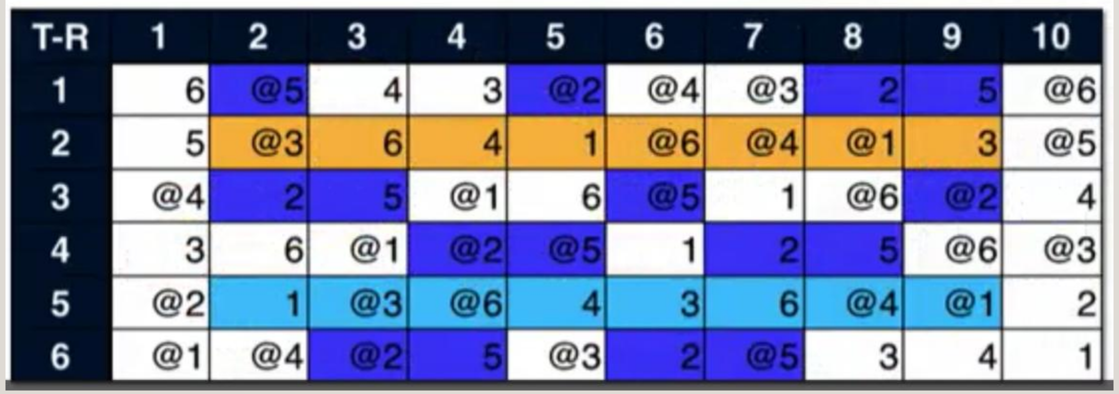
T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@3	@6	4	3	6	@4	@1	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	@2	1	5	2	@6	@3
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	@3	6	4	1	@6	@4	@1	3	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	2	1	5	@2	@6	@3
5	@2	1	@3	@6	4	3	6	@4	@1	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

OP 3: SWAP TEAMS (3) REPAIR ENTRIES

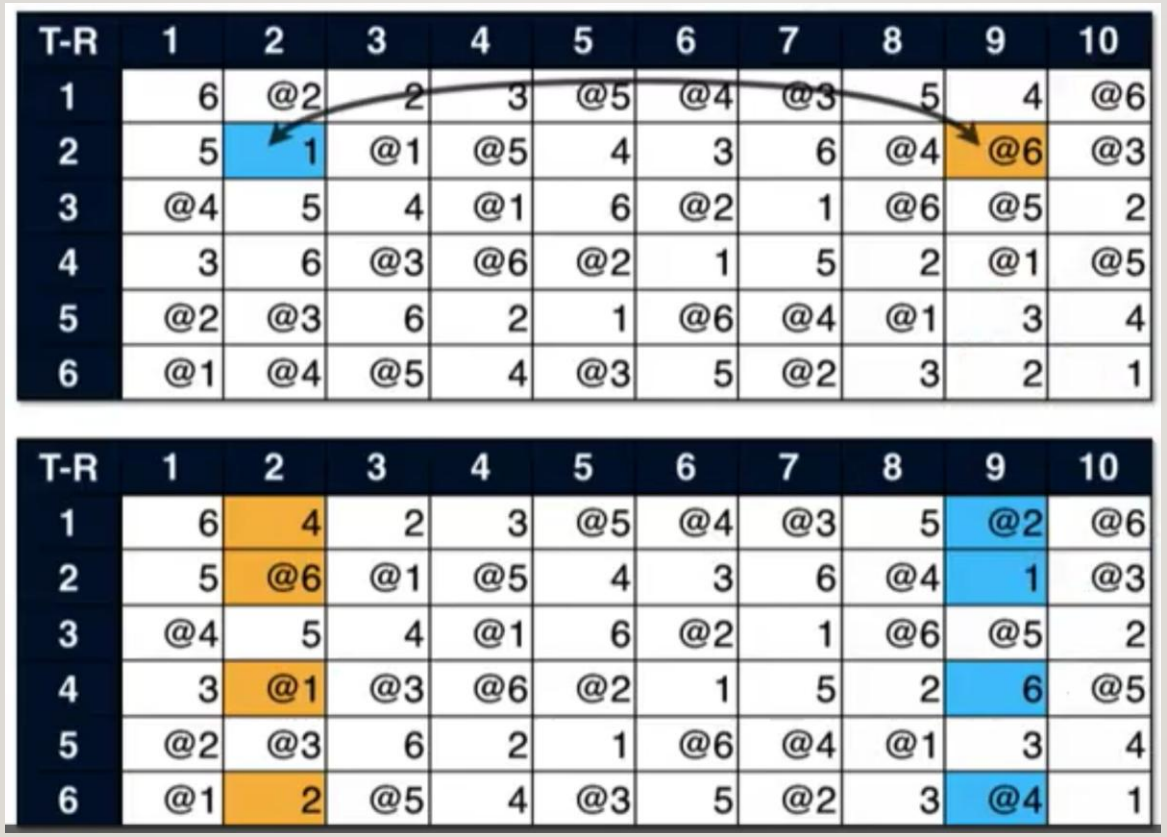


T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	@3	6	4	1	@6	@4	@1	3	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	2	1	5	@2	@6	@3
5	@2	1	@3	@6	4	3	6	@4	@1	2
6	@1	@4	@5	2	@3	5	@2	3	4	1



T-R	1	2	3	4	5	6	7	8	9	10
1	6	@5	4	3	@2	@4	@3	2	5	@6
2	5	@3	6	4	1	@6	@4	@1	3	@5
3	@4	2	5	@1	6	@5	1	@6	@2	4
4	3	6	@1	@2	@5	1	2	5	@6	@3
5	@2	1	@3	@6	4	3	6	@4	@1	2
6	@1	@4	@2	5	@3	2	@5	3	4	1

OP 4: SWAP PARTIAL ROUNDS

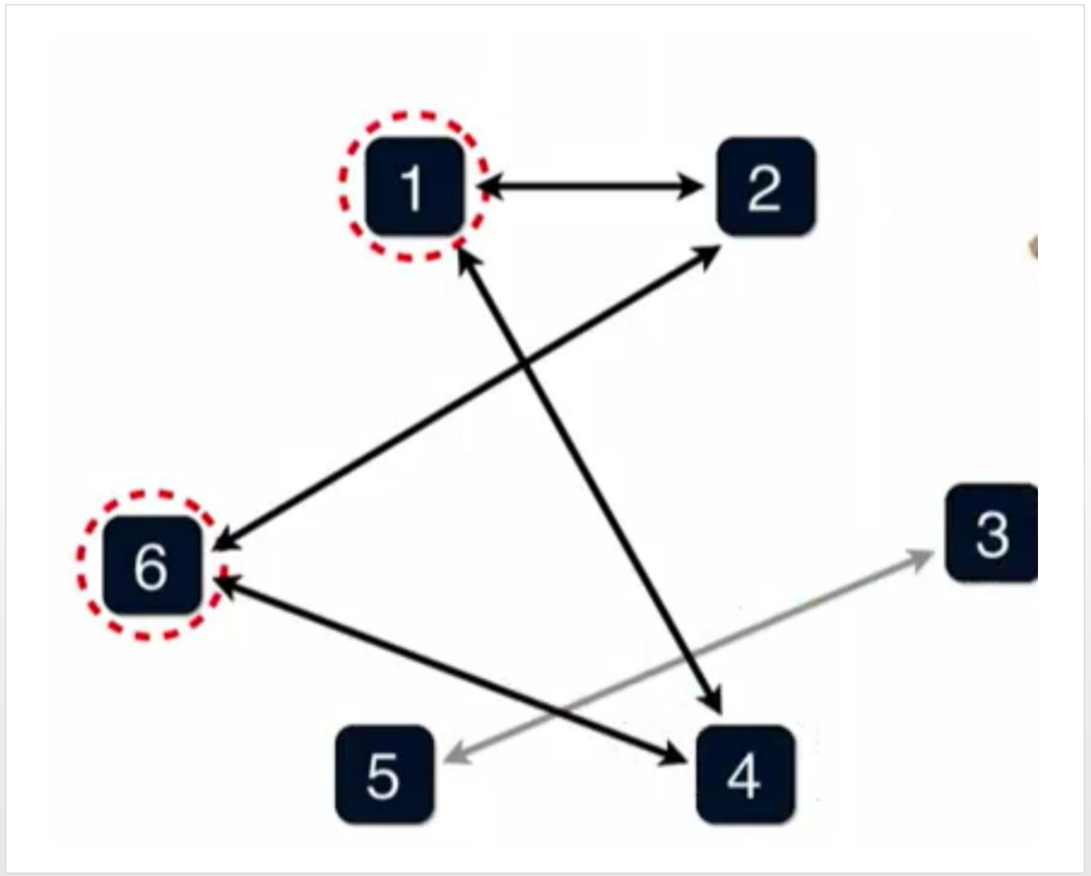


T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	2	3	@5	@4	@3	5	4	@6
2	5	1	@1	@5	4	3	6	@4	@6	@3
3	@4	5	4	@1	6	@2	1	@6	@5	2
4	3	6	@3	@6	@2	1	5	2	@1	@5
5	@2	@3	6	2	1	@6	@4	@1	3	4
6	@1	@4	@5	4	@3	5	@2	3	2	1

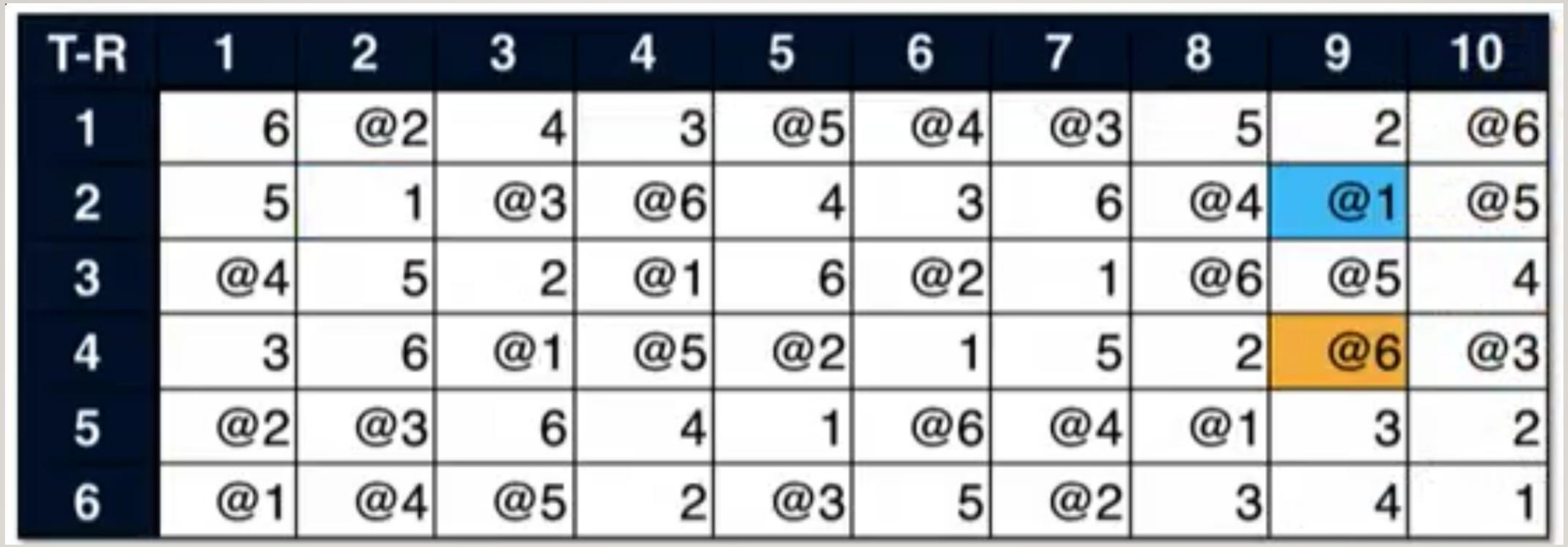
T-R	1	2	3	4	5	6	7	8	9	10
1	6	4	2	3	@5	@4	@3	5	@2	@6
2	5	@6	@1	@5	4	3	6	@4	1	@3
3	@4	5	4	@1	6	@2	1	@6	@5	2
4	3	@1	@3	@6	@2	1	5	2	6	@5
5	@2	@3	6	2	1	@6	@4	@1	3	4
6	@1	2	@5	4	@3	5	@2	3	@4	1

OP 4: SWAP PARTIAL ROUNDS

- **Red-dashed nodes (1 and 6)** are the two teams you initially swapped out of their rounds (Team 2's old opponent, Team 1, and Team 9's old opponent, Team 6).
- **Arrows** show "Team 1 was playing Team 2 in R2, so when you stole Team 2's slot, Team 1 got bumped → now Team 1 needs a new round." You then look up where Team 1 was originally in the table, and its arrow points there.
- You follow that arrow to **Team 4** (say Team 1 had been playing Team 4 in some Round X), bumping Team 4 next. Then you draw arrows from 4 to the team 4 was playing, and so on
- **The chain closes** when an arrow eventually points back to Team 6's original slot—the other red-dashed node—because that was the second "hole" you created at the start. Filling that slot completes the cycle, and all "@…" holes disappear.

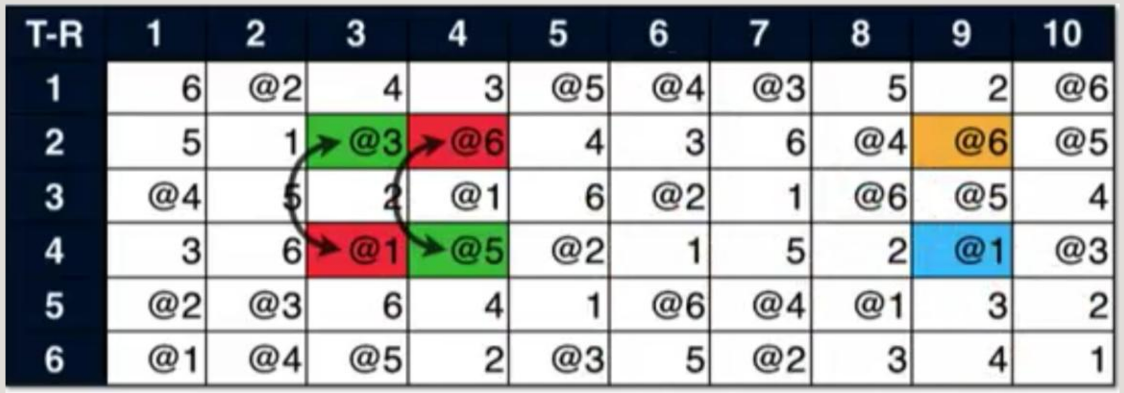


OP 5: SWAP PARTIAL TEAMS

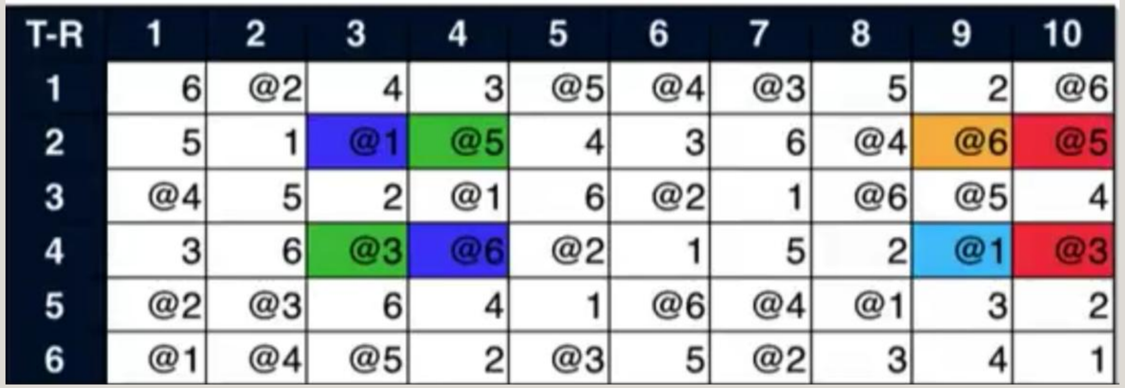


T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@3	@6	4	3	6	@4	@1	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	@2	1	5	2	@6	@3
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

OP 4: SWAP PARTIAL TEAMS REPAIR

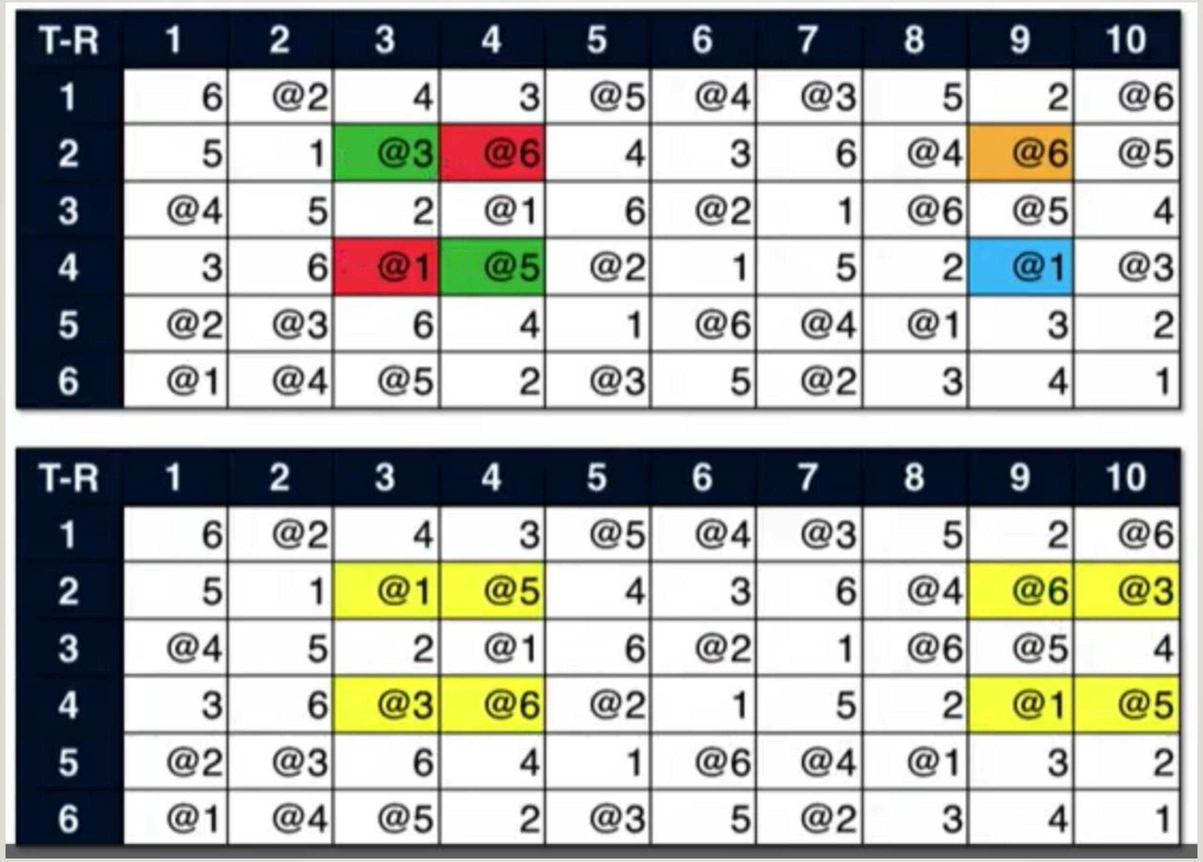


T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@3	@6	4	3	6	@4	@6	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	@2	1	5	2	@1	@3
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1



T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@1	@5	4	3	6	@4	@6	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@3	@6	@2	1	5	2	@1	@3
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

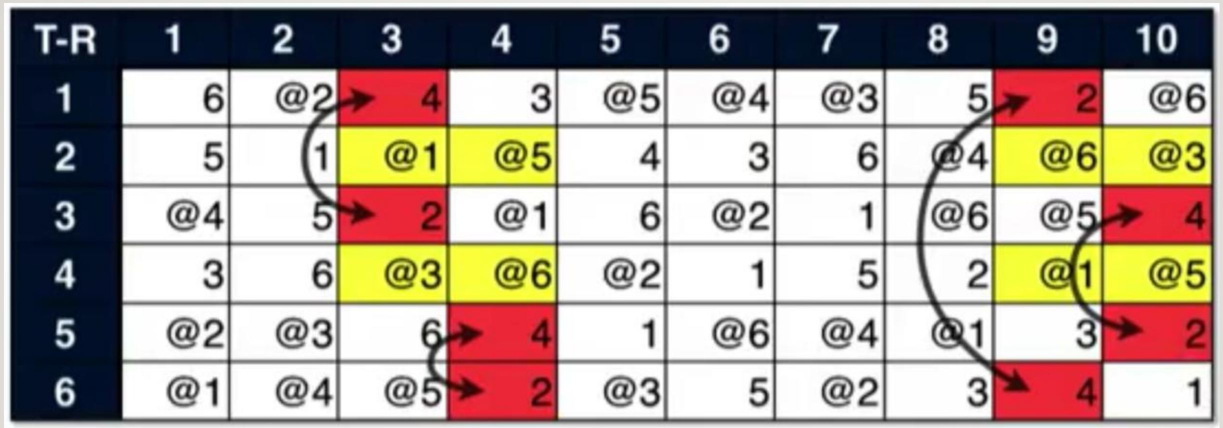
OP 5: SWAP PARTIAL TEAMS (2) PROPAGATE



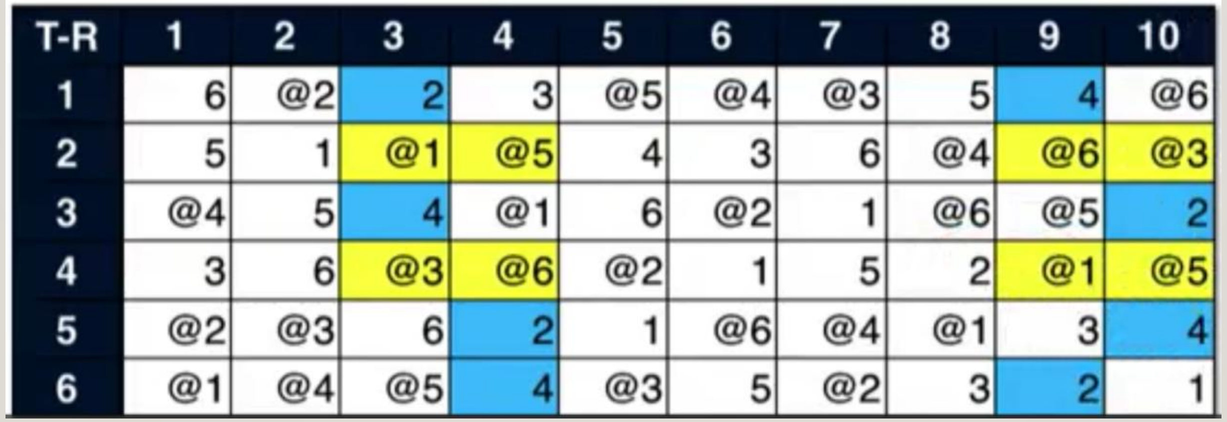
T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@3	@6	4	3	6	@4	@6	@5
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@1	@5	@2	1	5	2	@1	@3
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@1	@5	4	3	6	@4	@6	@3
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@3	@6	@2	1	5	2	@1	@5
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1

OP 5: PARTIAL SWAP TEAMS (FINAL)



T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	4	3	@5	@4	@3	5	2	@6
2	5	1	@1	@5	4	3	6	@4	@6	@3
3	@4	5	2	@1	6	@2	1	@6	@5	4
4	3	6	@3	@6	@2	1	5	2	@1	@5
5	@2	@3	6	4	1	@6	@4	@1	3	2
6	@1	@4	@5	2	@3	5	@2	3	4	1



T-R	1	2	3	4	5	6	7	8	9	10
1	6	@2	2	3	@5	@4	@3	5	4	@6
2	5	1	@1	@5	4	3	6	@4	@6	@3
3	@4	5	4	@1	6	@2	1	@6	@5	2
4	3	6	@3	@6	@2	1	5	2	@1	@5
5	@2	@3	6	2	1	@6	@4	@1	3	4
6	@1	@4	@5	4	@3	5	@2	3	2	1

BACK TO CVRP

SOLVING CVRP - GENERAL KEY POINTS

- No one solution
- Economic-Fast Objective/Heuristic Function
  - Precompute distances
  - Fast computation of tour length
- Compact Search Space
- Need to deal with Local Optima
  - (ILS, Mutation Control, Niching)

RECOMMENDATION

- Start with a baseline greedy algorithm e.g., nearest neighbor (can initially sort customers according to polar coordinates)

- Visualize the work of the heuristic – i.e., the resulting tour

OBJECTIVE OF ASSIGNMENT

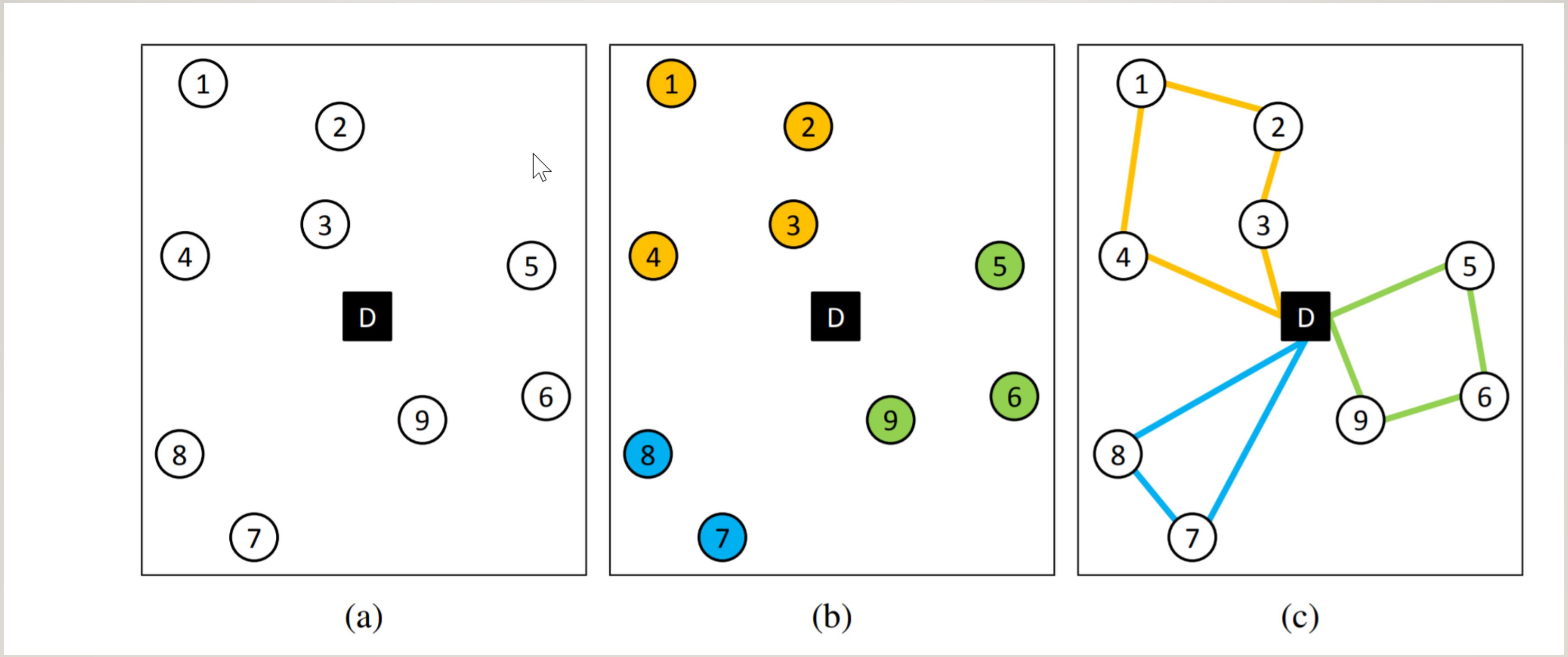
- Explore a different approach to CVRP
  - Try multiphase solution:
    - Solve the multi-knapsack using
      - New search strategy
      - Branch & Bound
    - Solve the TSP
      - Explore new heuristics
  - Two approaches:
    - Discrete
    - Simultaneous – Multi-Objective

HEURISTICS FAMILIES

1. Multi-Stage Heuristics
2. Construction Heuristics
3. Iterative Improvement Heuristics
4. Meta Heuristics

TWO-STEP HEURISTIC BASED ON MKP – MULTI KNAPSACK

TWO PHASED CVRP HEURISTIC



Overview of the CVRP and the 2-Phase-Heuristic. (a) Initial state with 9 customers and 1 depot. (b) Clustering phase results in three clusters found. (c) Routing phase determines shortest path inside each cluster

KNAPSACK MODELING

- Use: MKP, Fractional MKP

- Truck Capacity = Sack Capacity
- Demand = Weight of item
- Priority = Value of item
- Use:
  - Distance to warehouse
  - Distance to nearest neighbor to or
  - Average distance to neighboring customers

THE MULTI-KNAPSACK PROBLEM (MKP)

- An extension of the classic Knapsack Problem
- In the MKP, there are multiple knapsacks, each with its own capacity.
- The objective is to maximize the total value of items placed into the knapsacks without exceeding the capacity of any individual knapsack.

MKP DEFINITION

- **Items**: There is a set of n items, each with a value vi and a weight wi.
- **Knapsacks**: There are m knapsacks, each with a capacity Cj.
- **Objective**: Maximize the total value of the items placed in the knapsacks.
- **Constraints:** Each item can be placed in at most one knapsack.
- The total weight of the items in each knapsack must not exceed its capacity.

MKP + TSP FOR CVRP

1. Use MKP to assign customers to vehicles based on demand and vehicle capacity constraints.

- Ensure each vehicle's total assigned demand does not exceed its capacity.

2. For each vehicle, solve TSP to find the shortest route covering all assigned customers.
3. Combine the optimal routes of all vehicles to form the complete CVRP solution.

CONSTRUCTION HEURISTICS FOR SOLVING CVRP

INSERTION HEURISTICS

- In every step:
  - Option 1: Insert customers into partial tours **or**
  - Option 2: Combine sub-tours considering some capacities and costs to generate a complete solution

  - Option 1: Assign c1 to v1 or to v2
  - Assign c2 before after c1 in v1 or v2 etc.
  - Evaluate each configuration using distances
- **Variants:** include the cheapest insertion, farthest insertion, and nearest insertion.

CLARKE AND WRIGHT SAVINGS ALGORITHM

- A construction heuristic:

1. Construct a single tour for each customer
2. Calculate the saving that can be obtained by merging those single customer tours
3. Iteratively combine the best sub-tours until no saving can be obtained anymore

CLARKE AND WRIGHT ALGORITHM

1. **Initialization**: Begin with each customer served by a separate route directly from the depot.
2. For customers (i,j) the **savings** S_{ij} is obtained by: S_{ij} = d_{0i} + d_{0j} - d_{ij}
3. Sort Savings in descending order
4. Starting with the highest savings, attempt to merge routes if the merge does not violate the constraints.
5. **Termination**: The process continues until all possible merges have been considered. The result is a set of routes that collectively cover all customers while attempting to minimize the total distance traveled.

ITERATIVE REPAIR FOR CVRP

IMPROVEMENT HEURISTICS

- Try to iteratively enhance a given feasible
- solution, which is often generated by a construction heuristic
- A common methodology is to **replace or swap customers between sub-tours** taking capacity constraints into account

- Lin–Kernighan–Johnson, K-OPT

META HEURISTICS FOR SOLVING THE CVRP

USING META HEURISTICS

- Use the top-level strategies which guide local improvement operators to find a global solution

- ILS, Genetic algorithms, Tabu Search, Simulated annealing, ACO and Adaptive Large Neighborhood Search (ALNS)

GA FOR CVRP

- Parallel the LS methods -
- Local Gene Optimization:
  - Optimize individual genes (“memetic”) for better route order
- **Specialized Genetic Operators**:
  - **Special Crossover**: Choose best out (minimal route) of overlapping partial routes
  - **Special Mutation**: Relocate customer to minimize travel distance

ALNS FOR SOLVING CVRP

1. VRP SOLUTION REPRESENTATION

- A solution is represented as a sequence of customers visited by each vehicle

- For example:
  - for three customers (A, B, and C) and two vehicles (V1 and V2):
  - a possible solution could be: V1: [A, B], V2: [C]

2. NEIGHBORHOOD STRUCTURES

- Three operators are defined for this problem:
- Given: V1: [A, B], V2: [C]

1. **Relocate**: Move a customer from one route to another, e.g., V1: [A], V2: [C, B]
2. **Swap**: Swap two customers between two routes, e.g., V1: [A, C], V2: [B]
3. **Cross-Exchange**: Exchange subsequences of customers between two routes (while respecting capacity constraints)
4. **Two-Opt**: Reverse the order of customers within a route, e.g., V1: [B, A], V2: [C]

3. VRP ADAPTIVE SEARCH STRATEGY

In the beginning, all neighborhood operators have the same probability of being selected

As the algorithm progresses, the probabilities are updated based on the success of each operator in improving the solution

4. VRP ACCEPTANCE CRITERION

Example: use the Simulated Annealing (SA) acceptance rule (“Metropolis step”)

Initially, worse solutions have a higher probability of being accepted, but this probability decreases over time, allowing the algorithm to converge to an optimal solution

5. STOPPING CONDITION

- The algorithm is terminated after a certain number of iterations or a time limit or a convergence criteria