AI LAB

Artificial Intelligence LAB
Shay Bushinsky, Fall 2021

LAB4: Discrete Optimization

Linear programming relaxation

MULTI OBJECTIVE FUNCTIONS

Multi objective functions

- Multi-objective optimization problems involving two or more objectives
- Optimal decisions need to be taken in the presence of trade-offs between conflicting objectives
- Ex: buying a car - minimizing cost while maximizing comfort

Pareto Optimal

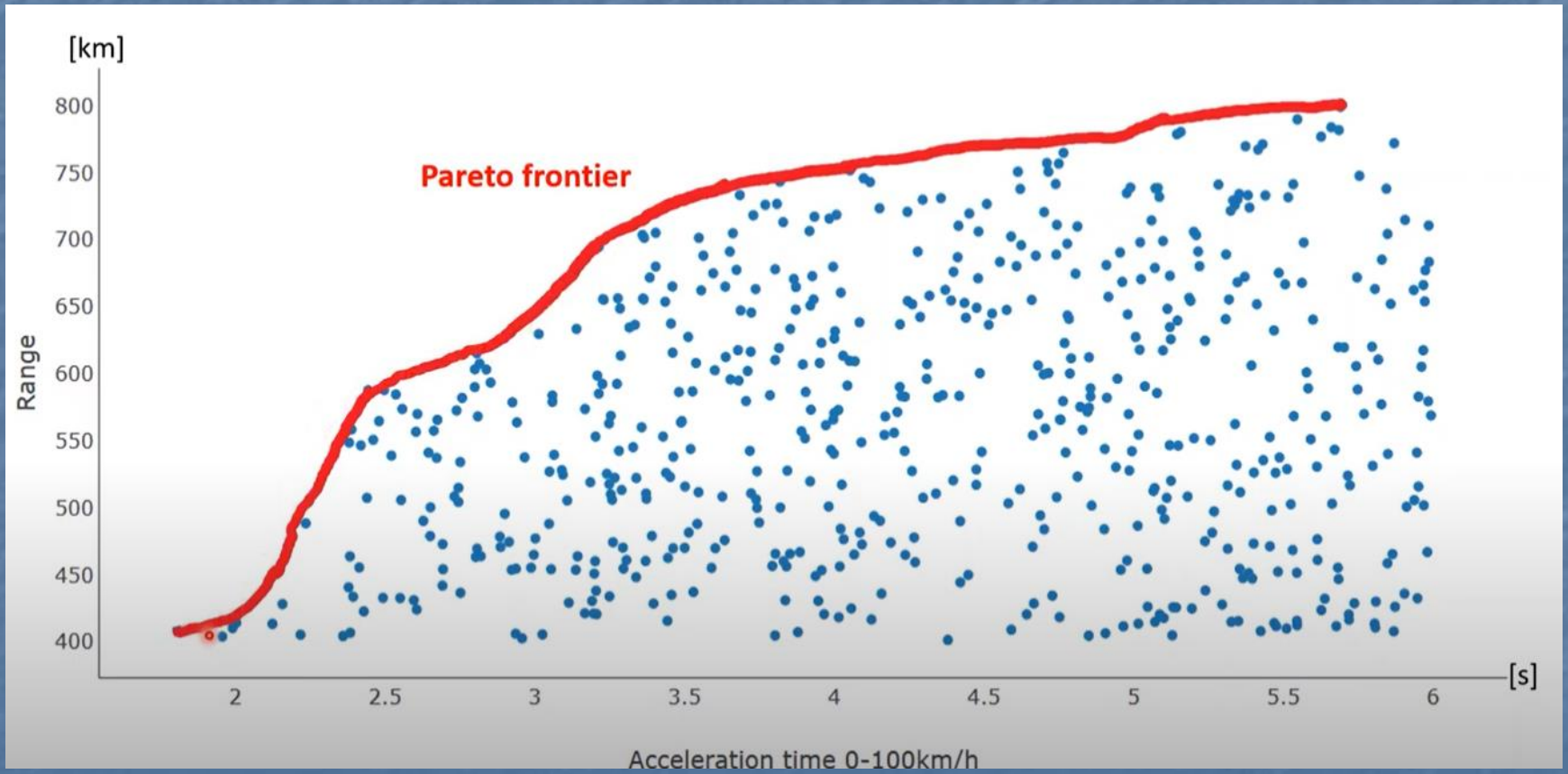
- For a nontrivial multi-objective optimization problem, there does not exist a single solution that simultaneously optimizes each objective
- In such case, the objective functions are said to be conflicting, and there exists a (possibly infinite) number of "Pareto optimal" solutions

GA Perspective

- Ex: Building an electric car - conflicting objectives:
  1. Increase acceleration time
  2. reduces the range and vice versa
- A gene: [wheel size, engine power, battery capacity]

- There is no one perfect solution

Pareto Frontier: max range, min acceleration time



Pareto Optimal Definition

- A solution is called Pareto Optimal, if none of the objective functions can be improved in value **without degrading** some of the other objective values
- Without additional subjective preference information, all Pareto optimal solutions are considered **equally good** (as vectors cannot be ordered completely)

Optimization Techniques

1. The Weighted Sum approach
2. A Probabilistic sampling approach

Problem Example

- Suppose one wants to minimize both of the following functions simultaneously:

- F1 = 750x1+60(25-x1) x2+45(25- x1)(25- x2)
- F2 = (25- x1) x2

Weighted Sum Approach

Convert a multiple objective problem into a single objective problem:

1. Assign a weight to each function such that:

$$w_1 + w_2 = 1 \text{ and } w_1 , w_2 \geq 0$$

1. Solve: FT = $\sum w_i$ Fi = w_1 F1 + w_2 F2

But this will only provide us with a single Pareto point…

Finding all points: Alter the weights and solve again

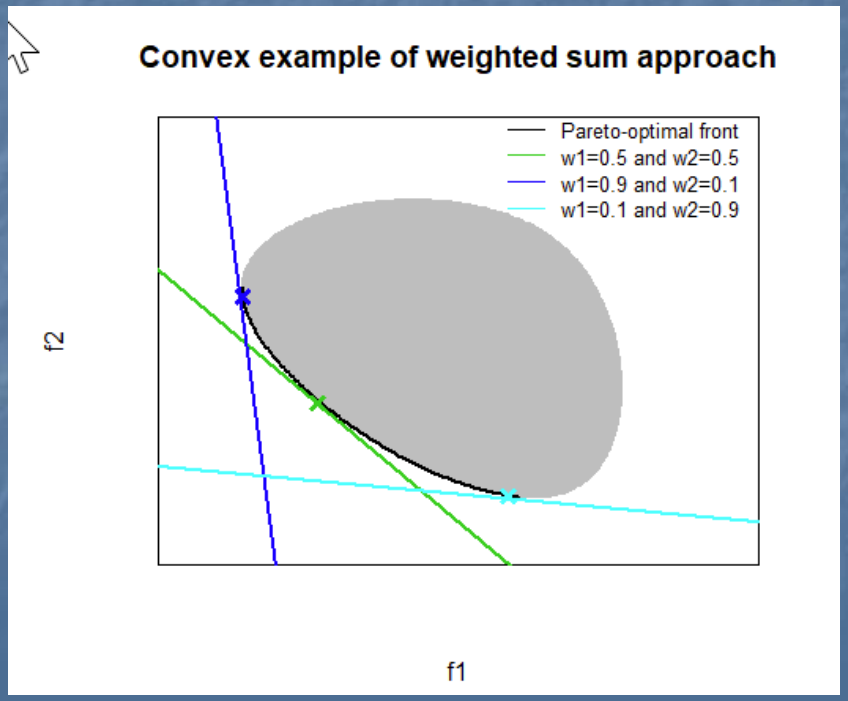
Selecting the Weights

1. The weight given to an objective is normally assigned to be **proportional** to the objective's **relative importance** in our problem
2. It is also required to account for the **magnitude** of our objective functions

- Resolution: scale all of the objective functions to the same magnitude (normalization)

Gradient methods

- The gradient of the slope is calculated by −w1/w2 applicable to convex set of pareto optimal solutions



Weighted Sum approach typical Problems

1. Inability to generate points in non-convex portions of the frontier
2. Inability to generate a uniform sampling of the frontier
3. A non-intuitive relationship between combinatorial parameters (weights, etc.) and performances
4. Poor efficiency (can require an excessive number of function evaluations)

Alternative approach

- Find the optimal weights using GA
- Using the combined function as a fitness function and the weights as genes

NSGA-II

- A Fast and Elitist Multi-objective Genetic Algorithm
  - Deb, K., Pratap, A., and Agarwal, S.. A
- **N**on-dominated **S**orting **G**enetic **A**lgorithm
- For multi-objective function optimization

Method

- Successive sampling of the search space
- The best individuals are calculated by non-dominated sorting breaking ties using the crowding distance

The Population

- Population of parameters:

- R_t : the new population
  is composed out of two populations:

1. P_t the previous population
2. Q_t the offspring's of P_t
3. P_{t+1} is the next generation

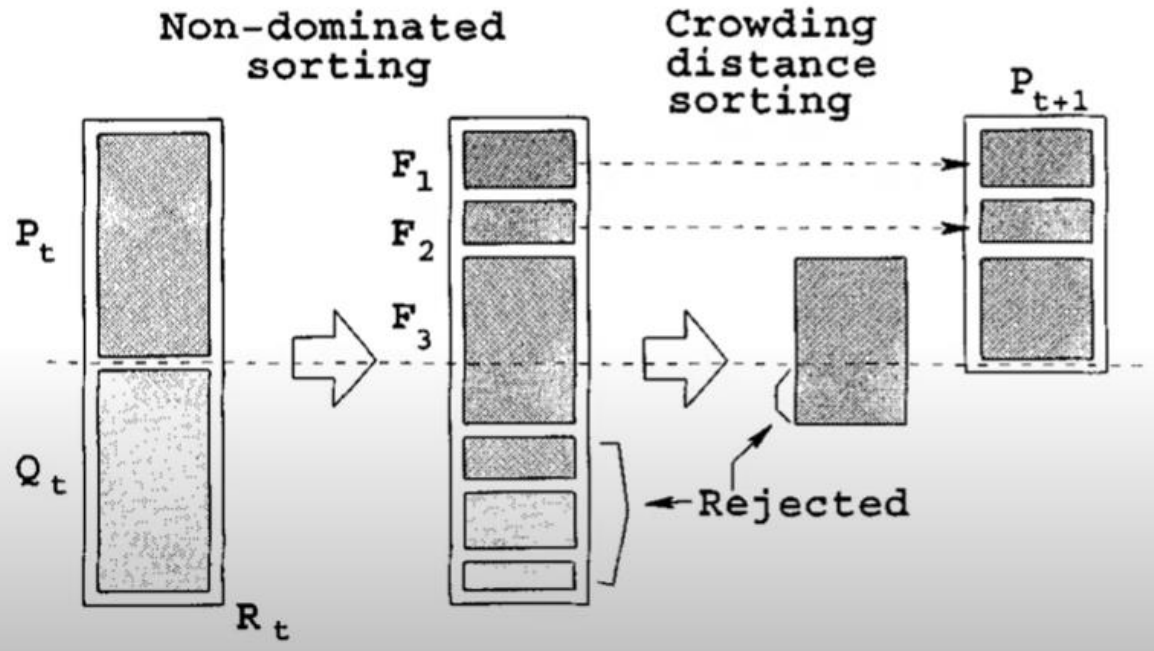
High Level algorithm

- While termination criteria == False:
  - Assume we have P_t and its offspring Q_t populations
  - Rank $P_t + Q_t$ by front – major index
  - Rank $P_t + Q_t$ by crowding distance – minor index
  - Sort $P_t + Q_t$
  - Select top of sorted generating: P_{t+1}
  - Generate Q_{t+1}

Population Ranking

- Ranking on different fronts:
- every individual is ranked according to the objectives (target indicators)
- Several fronts are maintained: f1, f2 ,f3...
- The smaller the front index - the better the front is in terms of its gene performance
- Fronts are added until the entire population is filled
- (the last front may not be inserted at its entirety)

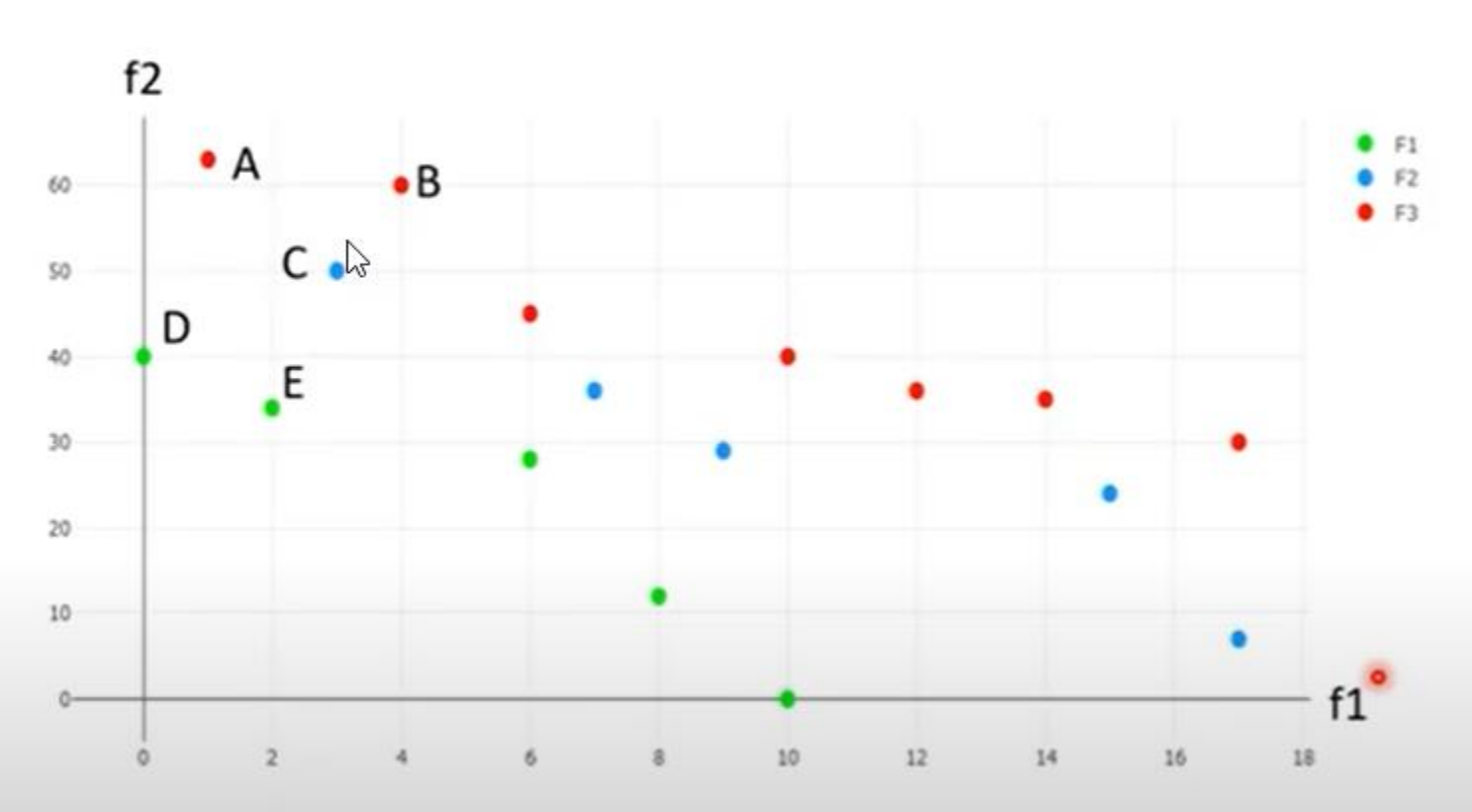
NSGA-II Main-Loop



K. Deb, IEEE Transtions On Evolutionary Computation Vol 6, No 2,April 2002

1. Create offspring and a combined population R_t
2. Rank and sort offspring due to performance on defined target indicators
3. Take best members to create new population including a good spread in solutions

Non-Dominated Sorting



Remark: don’t confuse f1 and f2 on the axes which are the performance indicators and the colored F1, F2, F3 which are the fronts…

Non-dominated Sorting

- Assign each gene to its correct front
- Definition: An individual **dominates** another individual iff it is better than the other in at least one indicator and not worse on all the other indicator(s)

Domination Example

- Assume the objective is to minimize indicator1 and indicator2 (two kpis):

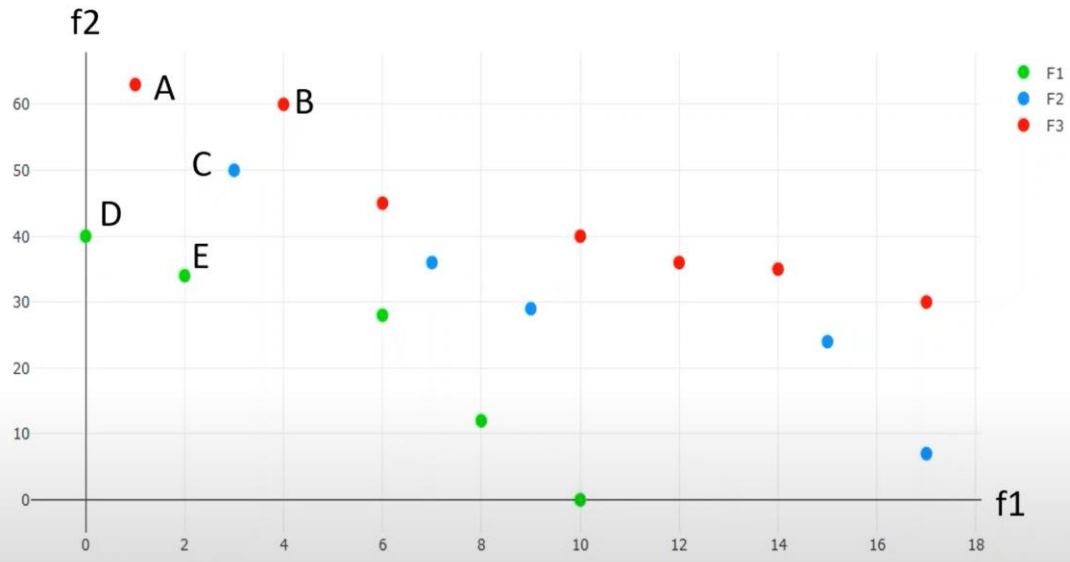
- g1 = (x1,y1) , g2=(x2,y2)
- g1 dominates g2 iff:
- (x1<=x2 & y1<=y2) & (x1<x2 | y1 < y2)

non-dominated sorting algorithm

- Each gene ("current") is compared with all the others for domination
- Two elements are generated:

1. a **list** of all individuals **dominated by** the current individual
2. a **counter** (domination_count) set to how many individuals **dominate** the current individual

Example



$A(x1|y1)$ dominates $B(x2|y2)$ when:

$$(x1 <= x2 \; and \; y1 <= y2) \; and \; (x1 < x2 \; or \; y1 < y2)$$

Individual C:
Domination_count = 2
Dominates = [B]

Individual D:
Domination_count = 0
Dominates = [A,B,C,..]

Individual E:
Domination_count = 0
Dominates = [B,C,..]

Generating Fronts

a) The first front (f1) will include all individuals with domination_count =0 meaning none of f1 members have an individual that dominates them

b) Scan all members of f1 and subtract 1 from all its list members domination_count

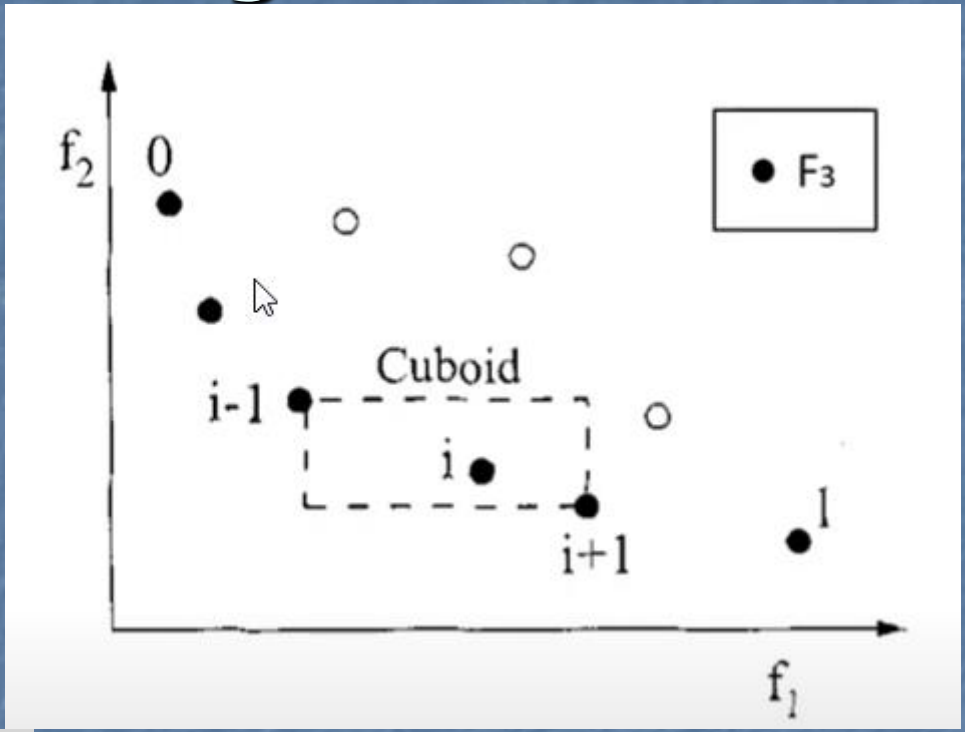
Reiterate steps a and b to create fronts f2, f3 etc.

secondary index generation: prepare to create offsprings:

- Use **crowding-distance** for the selection-strategy to avoid local optima:

- Individuals with higher crowding distance are picked first
- The total crowding distance for each gene is the sum of its crowding-distances per target-indicator (objective)

Crowding Distance Sorting



For o in objectives:

$$sort(front, o)$$
$$distance(front(\min)) = \inf$$
$$distance(front(\max)) = \inf$$
$$distance(i) = distance(i) + \frac{o(i+1) - o(i-1)}{o(max) - o(min)}$$

Algorithm for computing the distance of individual i:

- for o in objectives:
- sort(front, o) # sort front per indicator o
- distance(front(min)) = inf
- distance(front(max)) = inf
- distance(i) = distance(i) + ((o(i+1) - o(i-1)) / (o(max) - o(min))
- this means that the delta of the point from its neighboring points is scaled according to the delta of the extreme points in the front

process C: create offspring

- Repeat:
- 1. Tournament Selection:
- select parent1: (niche criteria)
- Sample g1 and g2
- g1=(a1,b1) where a1 is its rank and b2 is its crowding distance
- g2=(a2,b2)
- e.g. if a1=a2 & b1 > b2 select g1 as parent for reproduction
- select parent2: (using the same process)
- 2. Perform Crossover between parent1 & parent2
- 3. Mutation

Initialization

- Initially a random parent population P_0 is created
- This population is sorted based on non-domination
- Each solution is assigned a fitness equal to its non-domination level (1 is the best level)
- Thus, minimization of fitness is assumed
- Binary tournament selection, recombination and mutation operators are used to create child population Q_0 of size N
- Then, From now on:

Generation Iteration

- $R_t = P_t \cup Q_t$
  # combine parent and children population (R_t will be of size 2N)

- F = fast-nondominated-sort(R_t)
  # F = (F1; F2; : : :), all non-dominated R_t fronts

-

Generation Iteration cont.

- until | P_{t+1} | < N :
  # till the parent population is filled
- crowding-distance-assignment(Fi)
  # calculate crowding distance in Fi
- $P_{t+1} = P_{t+1}$ U Fi
- # include i-th non-dominated front in the parent pop

Generation Iteration cont.

- Sort(P_{t+1}; >= n)
  # sort in descending order using >= n
- $P_{t+1} = P_{t+1}[0 : N]$
  # choose the first N elements of P_{t+1}
- Q_{t+1} = make-new-pop(P_{t+1})
  # use tournament selection, crossover and mutation to create a new population Q_{t+1}
- t = t + 1

Discrete Optimization

Linear programming relaxation

The 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.

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

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

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

Greedy heuristics

1. Smallest
2. Most valuable
3. Sorting by density
4. Combinations?

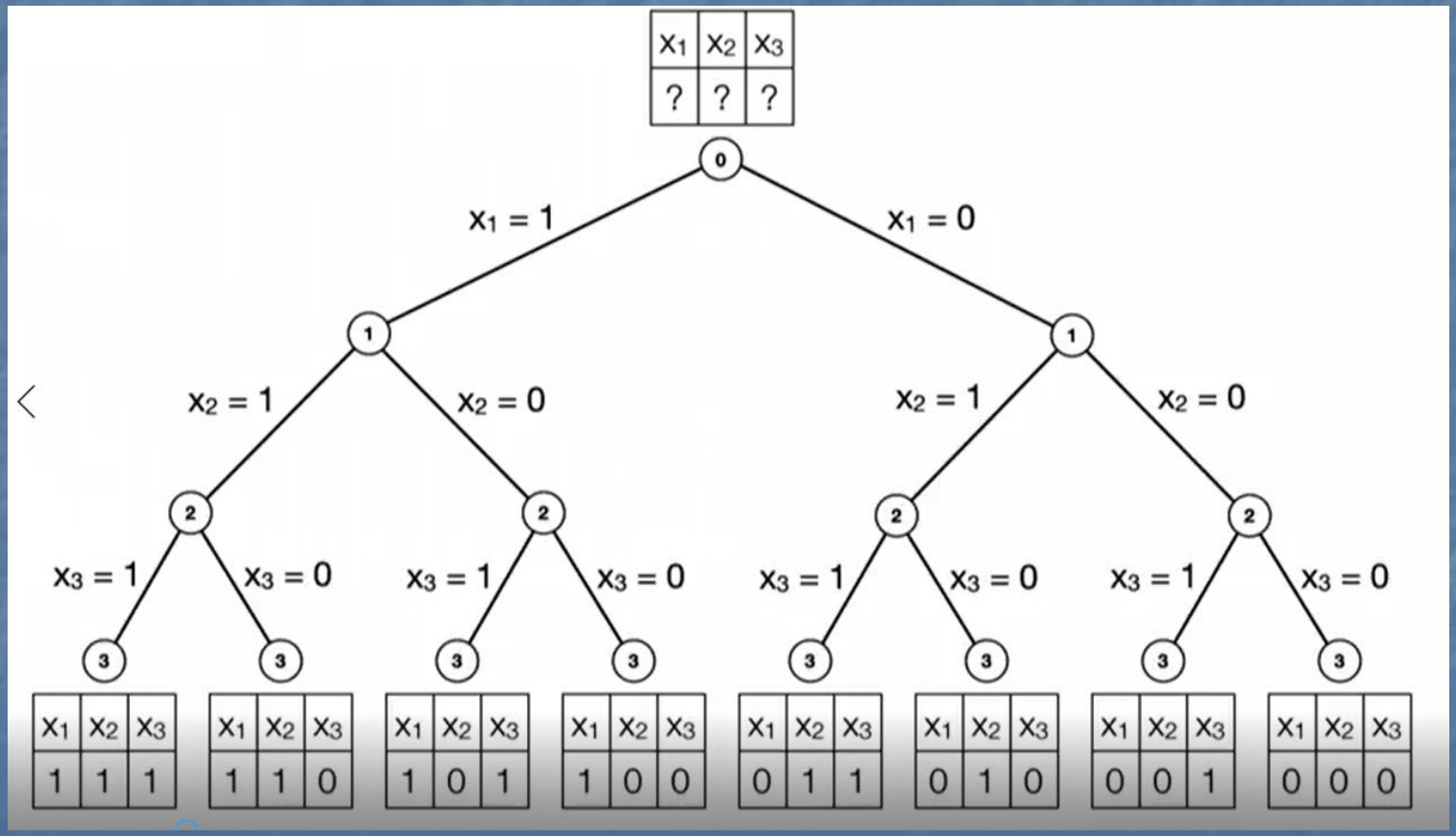
Easily feasible but is any of these optimal?

Key benefit: Obtaining a baseline but don't know how far from optimal…

The search space

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

Assigning the decision variables: Exhaustive search – decision tree



Branch and Bound

- Iterate between two steps –

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

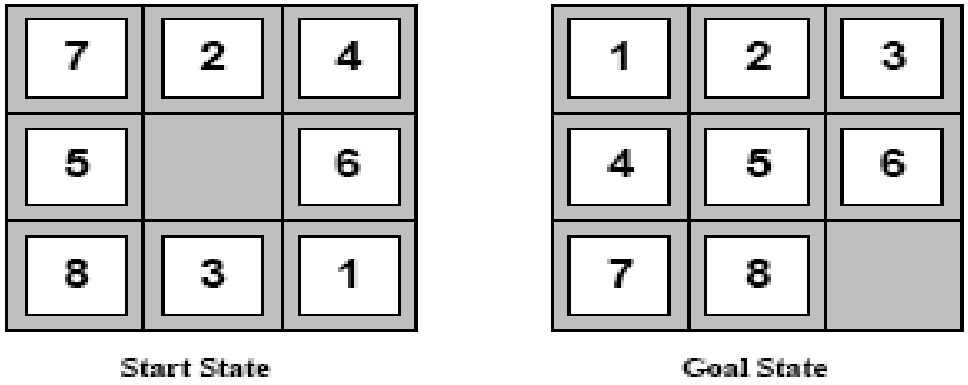
Relaxation

- 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

Relaxation examples: ignore blocking tiles: n-puzzle

$h_1(n)$ = number of misplaced tiles

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



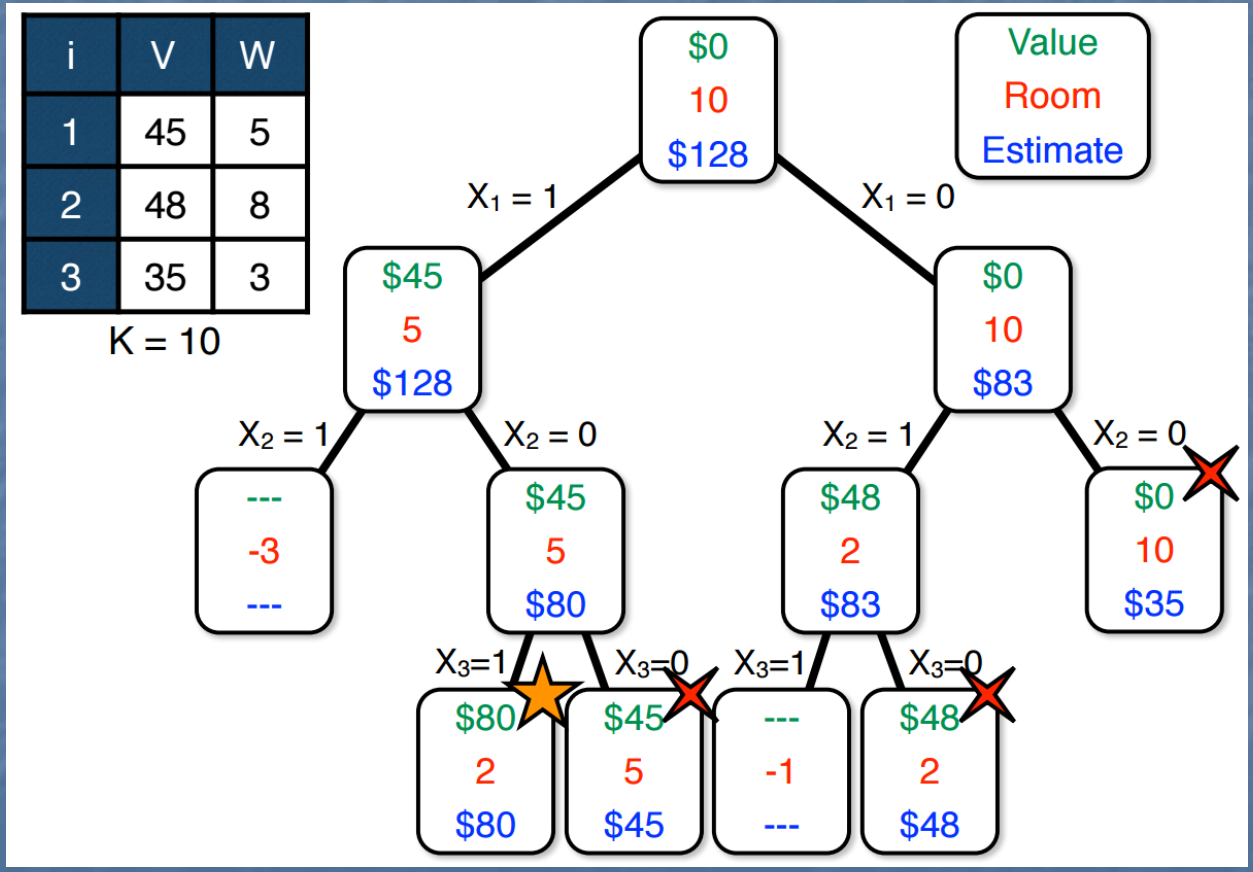
$h_1(S) =$?? 6

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

Knapsack problem

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

1. Relaxing Knapsack Capacity



2. Relaxing Integral item sizes

- Select the items while the capacity is not exhausted – select a fraction of the last item filling the knapsack
- Order the items by decreasing value of Vi/Wi – "most value per weight" - a notion of "density"

Option 2: neglecting integrality constraint

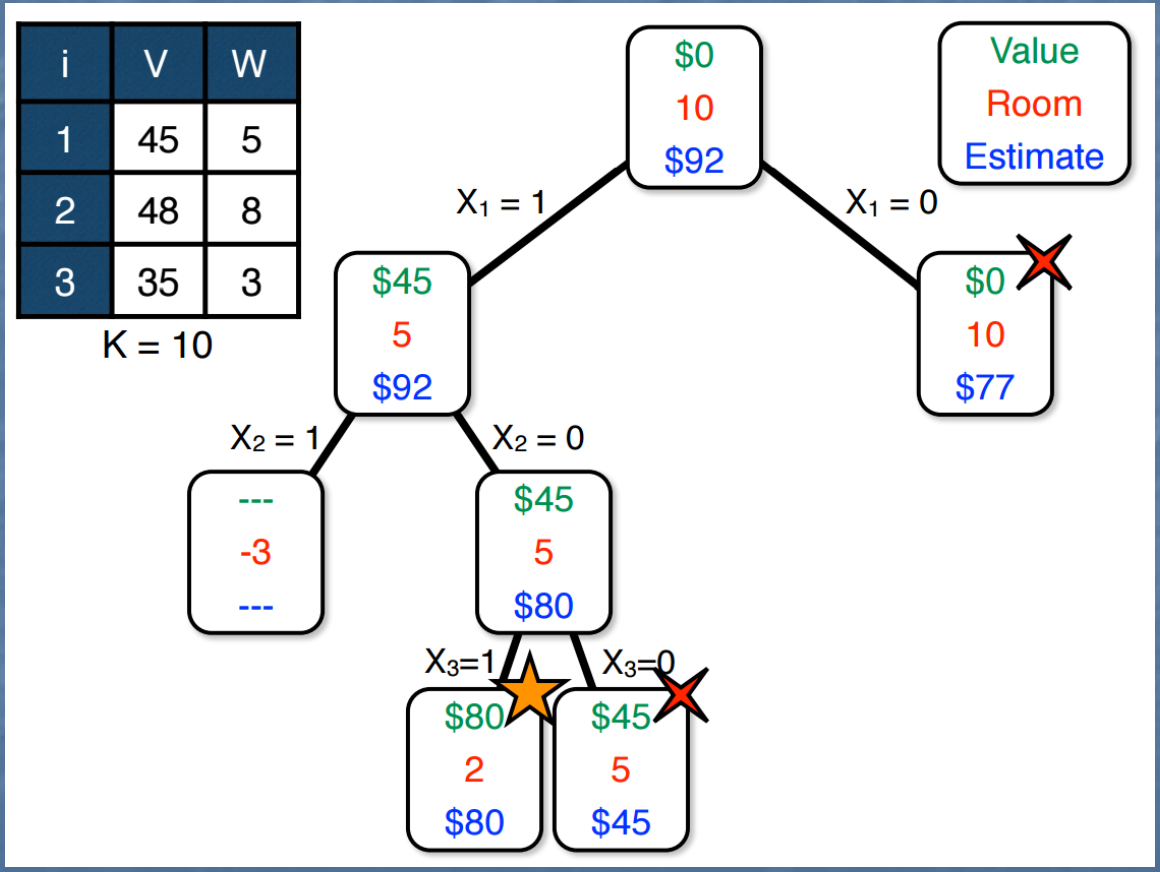
- Fractional heuristic: **0 <= Xi <= 1**
- look at Vi/Wi
- V1/W1=45/5=9
- V2/W2=48/8=6
- V3/W3=35/3=11.7
- Select Item 1 and 3 (the most valuable)
- Complete W with ¼ of item 2
- We obtain an initial bound (estimation) of 92 - an improvement over 128

Problem reformulation

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

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

Results in more aggressive pruning



Search Strategies

Algorithms

- Tradeoff time & space - Options

1. DFS (DLS) – linear- space effecient
2. Best First Search – heuristic dependent
3. LDS - hybrid

Search Strategies

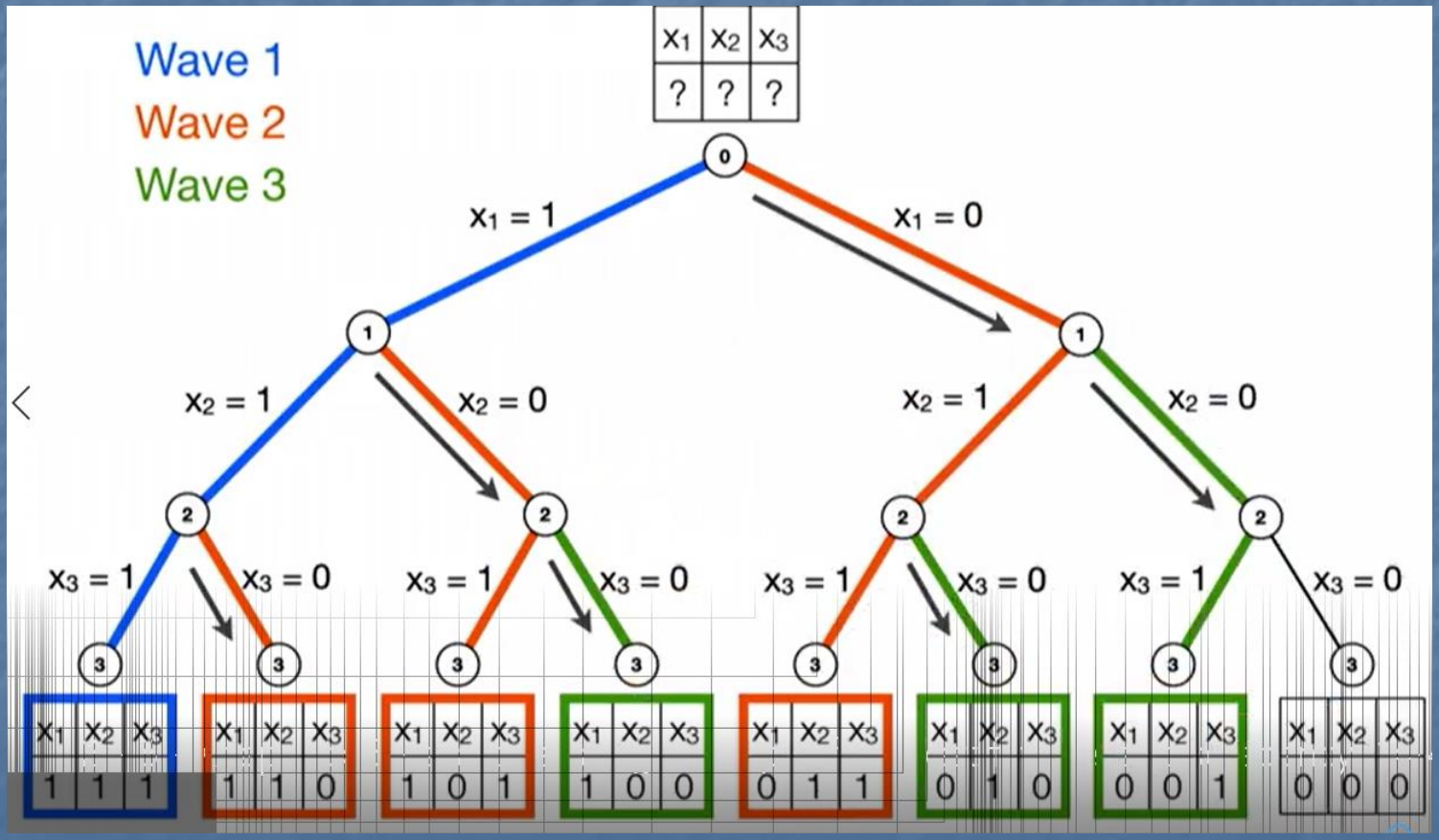
1. DFS – memory efficient (linear)

2. Best First Search – 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

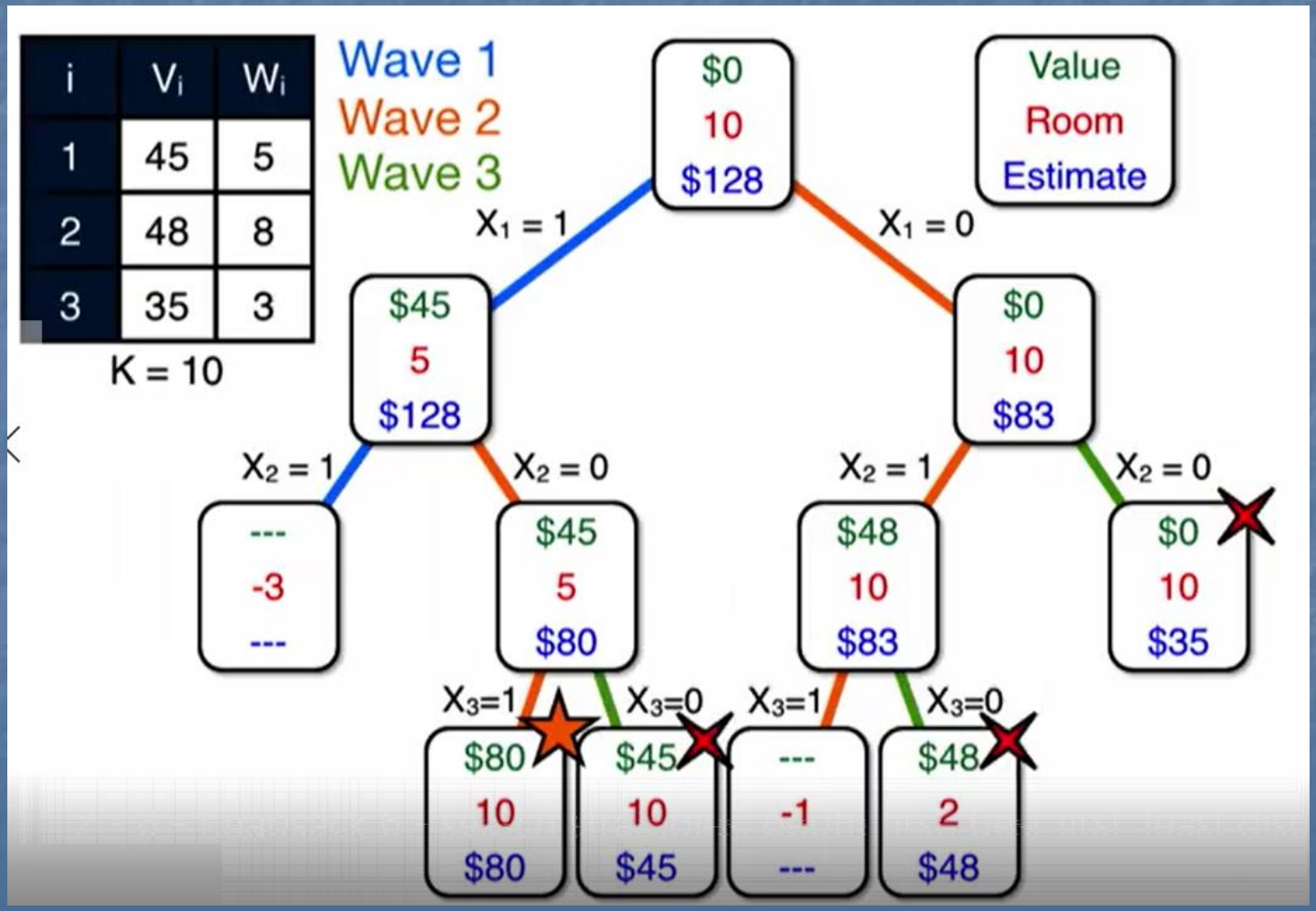
Least Discrepancy Search

- 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
x1 x2 x3
? ? ?
x1 = 1
x1 = 0
x2 = 1
x2 = 0
x2 = 1
x2 = 0
x3 = 1
x3 = 0
x3 = 1
x3 = 0
x3 = 1
x3 = 0
x3 = 1
x3 = 0
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



BACK TO CVRP

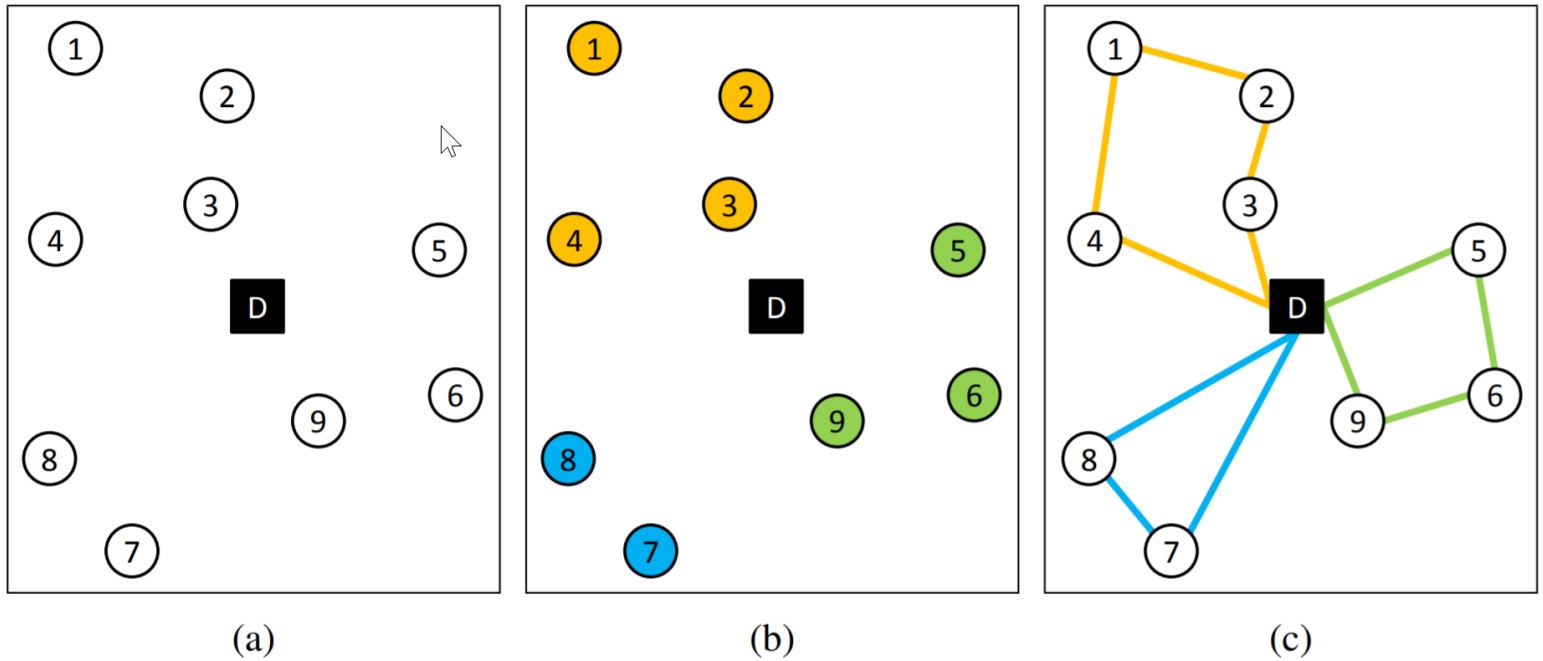
Objective of Assignment

- Explore a different approach to CVRP
  - Try multiphase solution:
    - solve the multi-knapsack
      - New search strategy: LDS
      - Branch & Bound
    - Solve the TSP
      - Explore new heuristics
      - Explore anti fast local optima convergence
  - Two approaches:
    - Discrete
    - Simultaneous - mutltiobjective

Heuristics Families

- construction heuristics
- improvement heuristics
- metaheuristics

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

Construction heuristics

- In every step:
  - insert customers into partial tours **or**
  - combine sub-tours considering some capacities and costs to generate a complete solution

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

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

Metaheuristics

- Can be thought of as top-level strategies which guide local improvement operators to find a global solution

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