# Ch 18: Branch-and-Bound

Goal: Find ALL optimal MILP solutions by partitioning the search space.

Branch: Make choices on certain variables.

Bound: Lower Bounds on solutions to give proof of optimality.

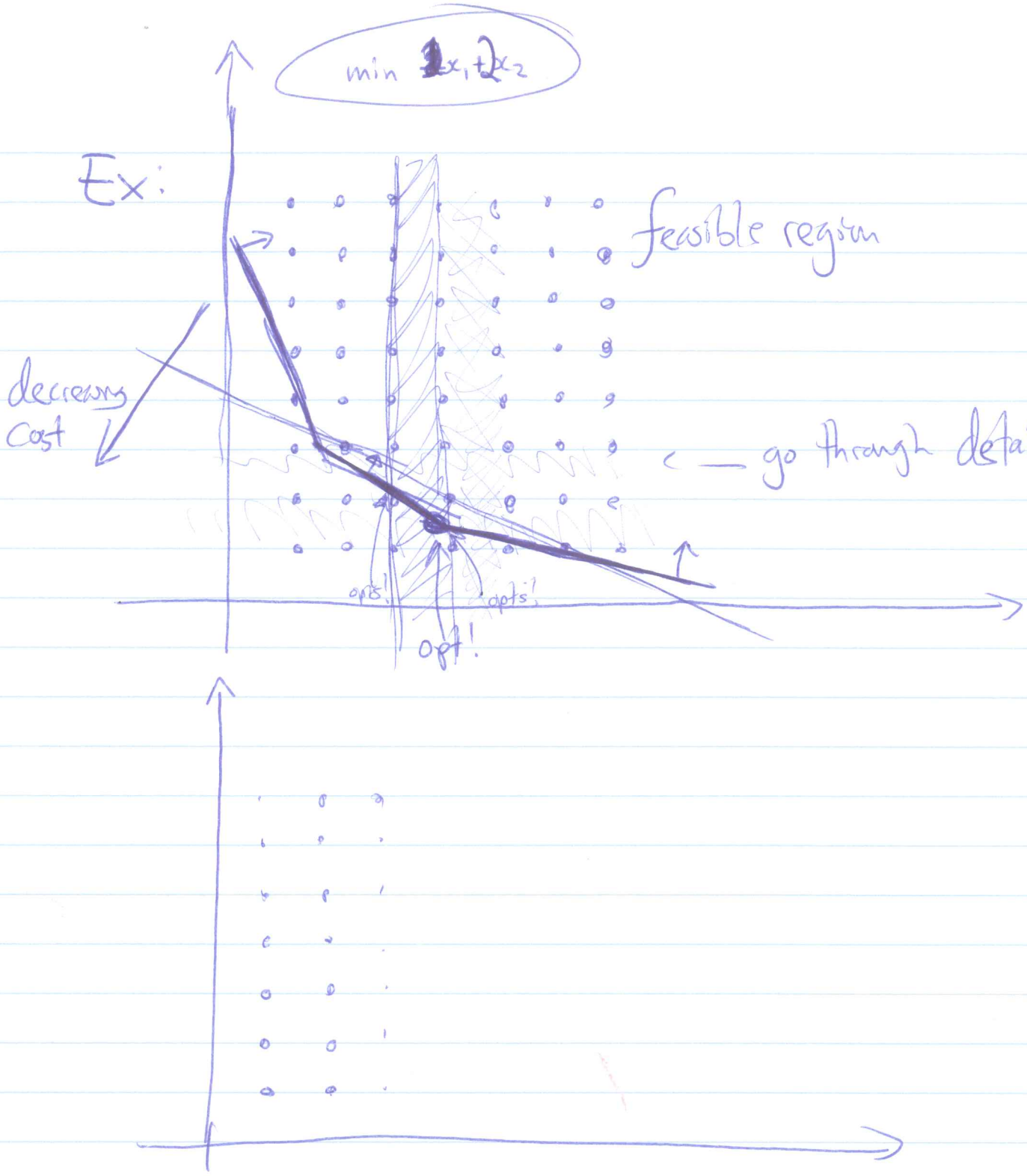
Consider $\min \underline{c}^T \underline{x}$
$A\underline{x} \leq \underline{b}$
$\underline{x} \geq \underline{0}$, integer.

Solve LP relaxation to find $\underline{x}^0$ optimum. Not necessarily an integer vector.
However, an optimal integer vector probably has integer values near these values....
So try 2 cases for non-integer (rounded) $x_i^0$:

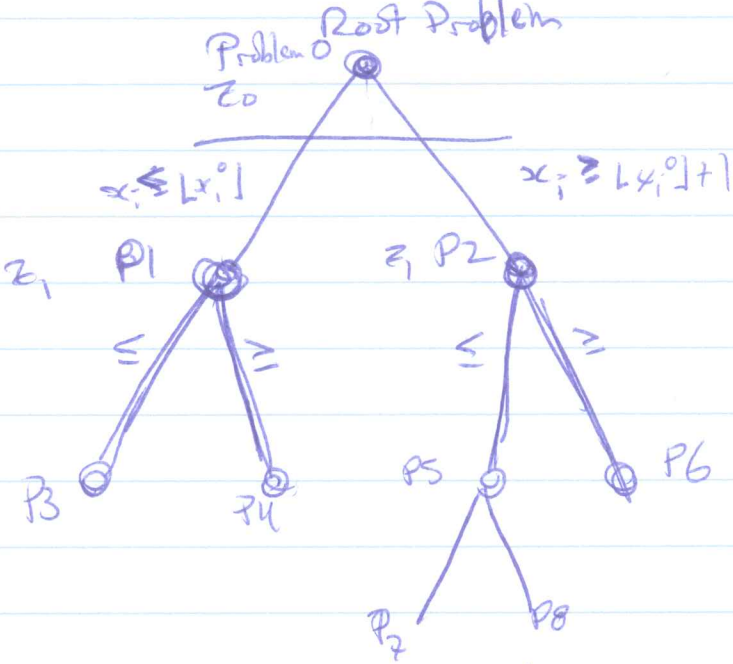
A: $\min \underline{c}^T \underline{x}$
$A\underline{x} \leq \underline{b}$
$\underline{x} \geq 0$ integer
$x_i \leq \lfloor x_i^0 \rfloor$

B: $\min \underline{c}^T \underline{x}$
$A\underline{x} \leq \underline{b}$
$\underline{x} \geq \underline{0}$ integer
$x_i \geq \lceil x_i^0 \rceil$

We can recursively solve each of these problems!


min 1x1 + 2x2
Ex:
feasible region
decreasing cost
← go through details!
opt!
opt!
opt!

# Branch-and-Bound Tree




$z_i$ = LP optimum for $P_i$

Things to track: Current best integer solution (& cost)

complete


~~Current best~~

If $z_i >$ current best integer solution, then $P_i$ is killed

We continue branching until all nodes are of these types:

1. Empty
2. Dead
3. Optimum is integer.

Flexibility: 1. Which NODE do we split?
2. How do we split it?

# General B-and-B.

(1) Branching.
(2) Lower Bounding.

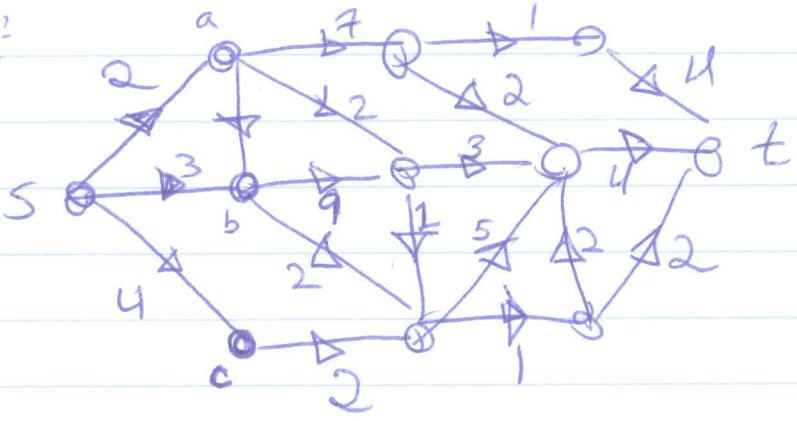
Branch And Bound ($P_0$).
    $z_0$ = lower bound of $P_0$.
    active set = $\{0\}$
    $U = \infty$
    current best = anything
    while active set $\neq \emptyset$ :
        Choose a branching node $k \in$ active set.
        Remove $k$ from active set.
        BRANCH: Generate children of $k$: $i_1, \ldots, i_t$.
        Add $i_1, \ldots, i_t$ to active set.
        LOWER: Compute lower bounds $z_{i_1}, \ldots, z_{i_t}$.
        For $j = 1, \ldots, t$ :
            If $z_{i_j} \geq U$ then kill child $i_j$.
            else if child $i_j$ is a complete solution then
                $U = z_{i_j}$, current best = child $i_j$. (ALSO: Kill nodes!)
            else
                Add child $i_j$ to active set.
    Output current best.

Example: Shortest st-Paths:

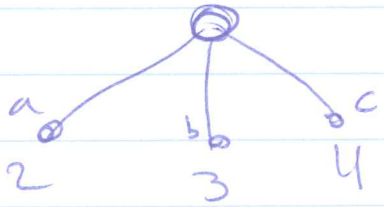
We consider every subproblem to be an sv-path and our branch is to pick the next step.

Once an st-path is reached, we can kill sv-paths that are longer.

Ex:



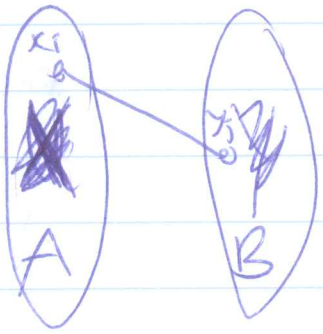

(dominance Relation)



# Branch & for TSP.

Simple: Branch on edge in-or-out.

Another: Use weighted Bipartite Matching. (Ch 10 & 11.)




Edge $a_i b_j$ corresponds to an edge $v_i v_j$

$wt(a_i, b_j) = cost(v_i, v_j)$.

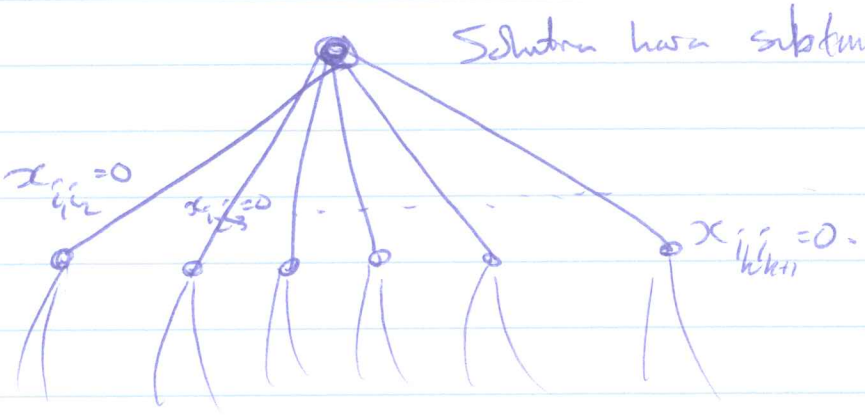
So, a min-weight perfect matching in this graph ≡ a set of vertex-disjoint cycles.

So Our branching can then take the following:

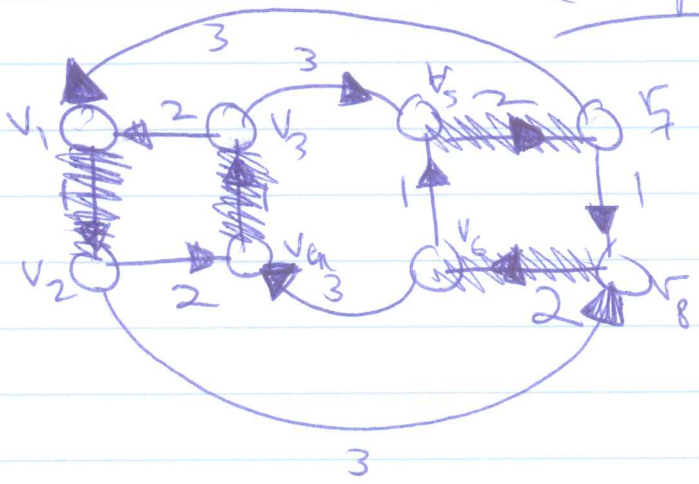
If $v_{i_1} v_{i_2} v_{i_3} \cdots v_{i_k}$ is a subtour, then

$x_{i_j, i_{j+1}} = 1$ for all $j \in \{1, \ldots, k\}$ $(v_{i_{k+1}} = v_{i_1})$

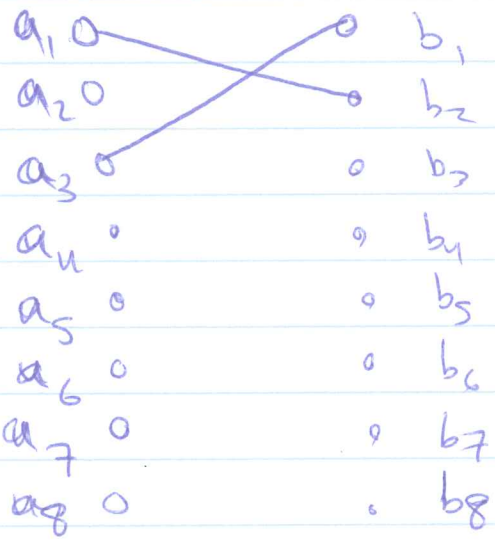
So branch on an edge that is removed!



# Branch-B for TSP Example:




← Presolving!



# Dynamic Programming

Solve sub-problems to help solve larger problems.

↳ "Last Decisions."

Principle of Optimality: If $D_1, \dots, D_m$ is an optimal sequence of decisions, then the decisions $D_i, \dots, D_m$ must be optimal.

Example: Consider a directed graph, where the vertices are partitioned into layers

$$\{s\} = V_0, V_1, \dots, V_t = \{t\} \qquad V(G) = \bigcup_{i=0}^{t} V_i$$

And all edges span $V_i$ to $V_{i+1}$ for some $i$.

Then, a shortest ($s$t) path can be seen as a sequence

$u_0 u_1 \dots u_t$ where each $u_i \in V_i$ and $u_i \to u_{i+1}$ is an edge and $\sum w(u_i, u_{i+1})$ is minimized.

(Decisions!)

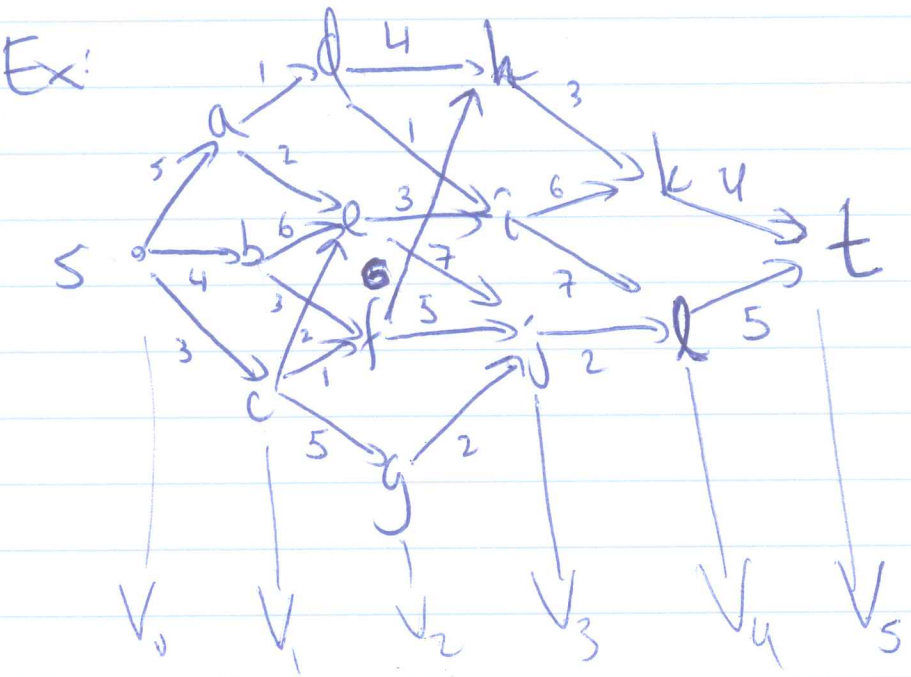
So if $u_0, \dots, u_t$ is optimal, then $u_i \dots u_t$ is optimal!

By carefully tracking the solutions to the sub-problems, we can solve the full problem.

# Dynamic Table

Table(i) = [ Node | $u_{i,1}$ | $u_{i,2}$ | ... | $u_{i,k_i}$
NextNode
Total Cost ]

Ex:




Table(4) = [ Node | k | l
Next | t | t
Cost | 4 | 4 ]

Table(3) = [ Node | h | i | j
Next | k | k | l
Cost | 7 | 10 | 7 ]

Table(2) =

| Node | d | e | f | g |
|---|---|---|---|---|
| Next | h | i | j | j |
| Cost | 11 | 13 | 12 | 9 |

Table(1) =

| Node | a | b | c |
|---|---|---|---|
| Next | d | f | f |
| Cost | 12 | 15 | 13 |

Table(0) =

| Node | s |
|---|---|
| Next | c |
| Cost | 16 |

Shortest Path: s, c, f, j, l, t.

---

For harder problems, the table can grow quite large!

Knapsack Problem:

I have a knapsack (backpack) that holds only a certain weight/volume.

I went home to my parents where they are offering me some of my childhood toys, each with a certain weight & sentimental value.

$$\max \sum v_i x_i$$

$$\text{s.t.} \quad \sum w_i x_i \le W \quad \leftarrow n_i \text{ (bounded)}$$

$$(x_i \le 1) \quad \leftarrow 0,1\text{-knapsack.}$$

$$x_1, \ldots, x_n \ge 0, \text{ integer}$$

Remove to Fractional.

Fractional Knapsack: "Liquid" items.

Strategy: Fill w/ most valuable until out or filled, move to next. ↑ (value/weight)

Unbounded, Integer Knapsack: (Mining)

Bounded, Integer Knapsack: $x_i \le n_i$

0,1 - Integer Knapsack: $x_i \le 1$.

---

Unbounded Knapsack w/ Dynamic Programming:

Decision: Put what in the bag (in what order?)

$I_1, I_2, \ldots, I_k \leftarrow k$ items.

After putting in item $i_1$, we have $W - w_i$ room left.

So $i_2 \ldots i_k$ is a solution to Knapsack w/ $W' \le W - w_i$.

Let $m[w]$ denote the max value of a weight-$w$ knapsack.

$m[0] \leftarrow 0$.
for $w$ in $\{1, \ldots, W\}$: (assuming integer weights)
$m[w] \leftarrow \max\{ v_i + m[w - w_i] : w_i \leq w \}$
$i[w] \leftarrow \text{argmax}\{ \ldots \}$

add a thing

maximize over what remains.

Ex: $\max\ 3x_1 + 3x_2 + 8x_3 + 16x_4$
s.t. $3x_1 + 2x_2 + 5x_3 + 8x_4 \leq 10$
$x_1, x_2, x_3, x_4 \geq 0$, integers.

$m[0] = 0$ $\quad i[0] = \emptyset$
$m[1] = 0$ $\quad i[1] = \emptyset$
$m[2] = 3$ $\quad i[2] = 2$
$m[3] = 3$ $\quad i[3] = \{1 \text{ or } 2\}$
$m[4] = 6$ $\quad i[4] = 2$
$m[5] = 8$ $\quad i[5] = 3$
$m[6] = 9$ $\quad i[6] = 2$

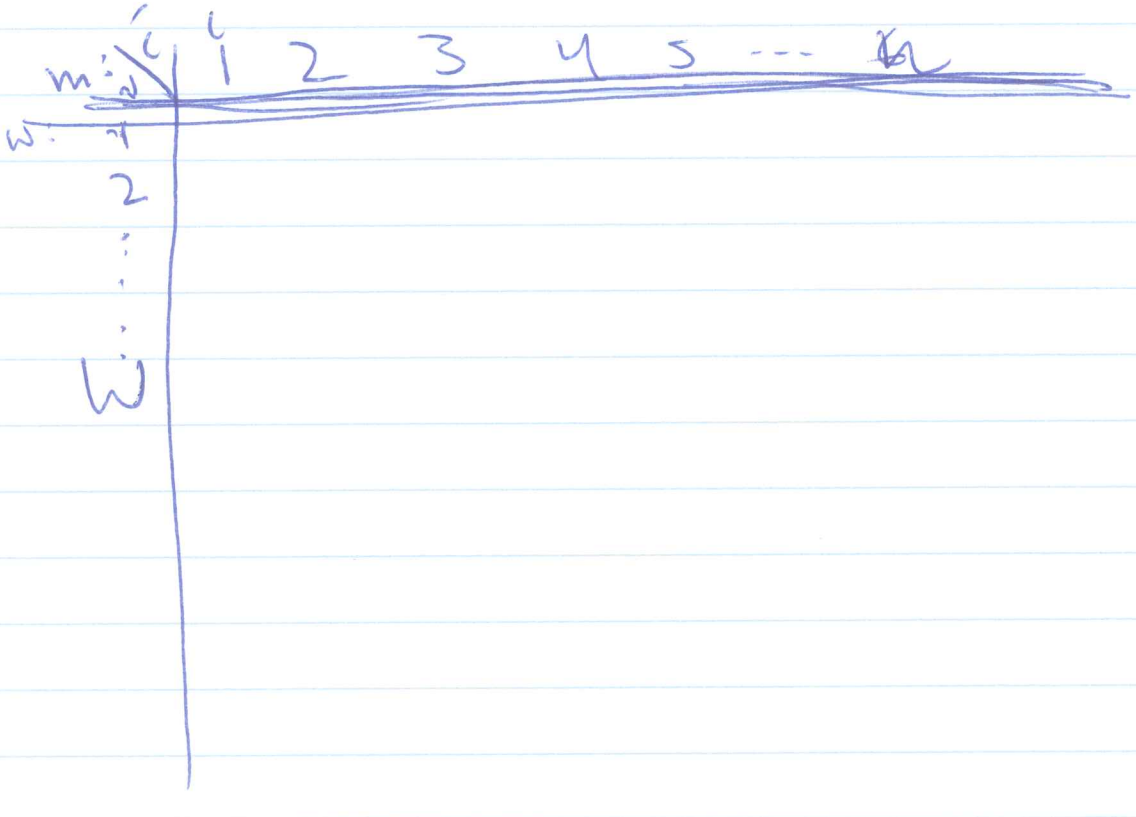
# 0,1 - Knapsack.

We need to track which items have been selected by the subproblems!

$m[i][w]$ = max knapsack w/ weight $\leq w$, using items $1, \ldots, i$.

$m[0][w] \leftarrow 0. \quad \forall w \in \{0, \ldots, W\}$.

```
for i ∈ {1, ..., n}:
    for w ∈ {1, ..., W}:
        if w ≥ w_i:
            m[i][w] ← max( m[i-1][w], m[i-1][w-w_i] + v_i )
        else:
            m[i][w] ← m[i-1][w]
```



Ex:

$$\max \quad 10x_1 + 5x_2 + 6x_3 + 20x_4$$
$$\text{s.t.} \quad 3x_1 + 2x_2 + 1x_3 + 4x_4 \leq 6$$
$$x_1, \; x_2, \; x_3, \; x_4 \in \{0,1\}.$$

Ex: $\max \quad 5x_1 + 6x_2 + 8x_3 + 11x_4$

$$2x_1 + 3x_2 + 3x_3 + 5x_4 \leq 9$$
$$x_1, \; x_2, \; x_3, \; x_4 \in \{0,1\}.$$

---

Local Search: For a feasible set $F$ & cost function $c$,

$N: F \longrightarrow 2^F$ neighborhood function.

$$\text{improve}(\underline{x}) = \begin{cases} \text{some } \underline{y} \in N(\underline{x}) \text{ with } c(\underline{y}) < c(\underline{x}) & \text{if it exists.} \\ \text{"No"} & \text{otherwise.} \end{cases}$$

Local Search {:

$\underline{x}$ = some feasible point in $F$ ← (1): How? Random?

while improve($\underline{x}$) ≠ "No": (2): What is $N$?

$\underline{x} \leftarrow$ improve($\underline{x}$) (3)

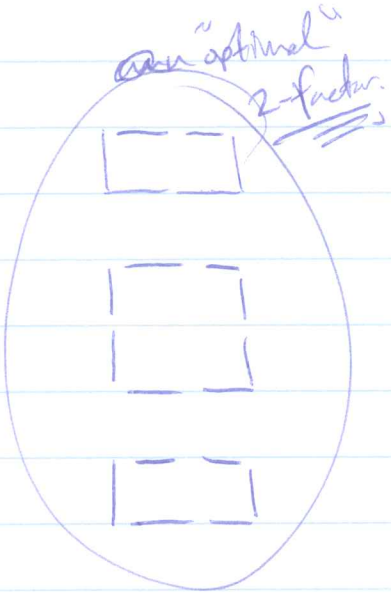
Output $\underline{x}$.

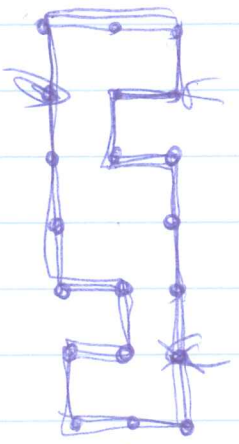
k-change neighborhood ($k \geq 2$) for TSP.

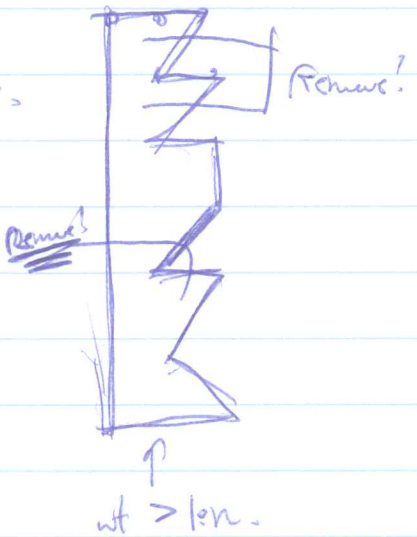
# Local Search for Graph Coloring.

Neighborhood options:

1. Replace color at $v$ w/ a lower available color.
2. Swap colors in a connected component of $G_{i,j}$


"optimal"
2-factor.


best.
wt: 1·n


Remove!
Remove!
wt > 1·n.