AI505

Optimization

Discrete Optimization

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

1. Integer Linear Programming

2. Dynamic Programming

Discrete Optimization

- Discrete optimization is a branch of optimization that deals with problems where the solution space is discrete, meaning that the variables can only take on specific, distinct values.
- This is in contrast to continuous optimization, where the variables can take on any value within a given range.
- Discrete optimization problems are often NP-hard, meaning that they are computationally challenging to solve.
- Discrete optimization is widely used in various fields, including operations research, computer science, engineering, and economics.
- It is important to develop efficient algorithms and heuristics to solve these problems, as they often arise in real-world applications.

Discrete Optimization

- What is discrete optimization?
- Problem formulation for linear programs
- Approximate solution techniques
- Exact solution techniques
- Dynamic programming

Discrete vs Combinatorial Optimization

In Combinatorial Optimization, variables have some combinatorial structure, ie, sets, permuations, paths, etc.

Definition (Combinatorial Optimization Problem (COP))

Input: Given a finite set $N = \{1, \ldots, n\}$ of objects,
weights c_j for all $j \in N$,
a collection of feasible subsets of N, $\mathcal{F}$

Task: Find a minimum weight feasible subset, ie,

$$\underset{S \subseteq N}{\text{minimize}} \left\{ \sum_{j \in S} c_j \mid S \in \mathcal{F} \right\}$$

COP can also be modelled as discrete optimization problems.

Typically: **incidence vector** of S, $\boldsymbol{x}^S \in \mathbb{B}^n$: $x_j^S = \begin{cases} 1 & \text{if } j \in S \\ 0 & \text{otherwise} \end{cases}$

Discrete (or Combinatorial) Optimization

Solving problems with variables that are discrete instead of continuous

Example

- Set covering and set partitioning
- Knapsack problem
- Traveling salesman problem
- Vehicle routing problem
- Job scheduling problem
- Facility location problem
- Bin packing problem
- Graph coloring problem
- Maximum flow problem
- Minimum spanning tree problem
- Shortest path problem
- Steiner tree problem
- Hamiltonian path problem

The set of possible discrete values can be finite or infinite

Solution Methods

Exact Methods:

- Integer programming
- Combinatorial optimization algorithms
- Graph theory algorithms
- Scheduling algorithms
- SAT
- Dedicated Branch and bound
- Dynamic programming
- Constraint programming
- Integer linear programming
- Mixed-integer programming
- Network flow algorithms

Heuristic Methods:

- Approximation algorithms
- Greedy algorithms
- Metaheuristics
  - Local Search algorithms
  - Genetic algorithms
  - Simulated annealing
  - Ant colony optimization
  - Tabu search

Outline

1. Integer Linear Programming

2. Dynamic Programming

Notation: Set of Integer Numbers

- $\mathbb{Z}$ set of integer numbers $\{..., -3, -2, -1, 0, 1, 2, 3, ...\}$
- $\mathbb{Z}^+$ set of positive integers
- $\mathbb{Z}_0^+$ set of nonnegative integers ($\{0\} \cup \mathbb{Z}^+$)
- $\mathbb{N}_0$ set of natural numbers, ie, nonnegative integers $\{0, 1, 2, 3, 4, ...\}$
- $\mathbb{B}$ set of binary numbers

Mixed Integer Linear Programming (ILP)

Linear Objective • Linear Constraints • but! integer variables

$$\begin{aligned} \max\ & \boldsymbol{c}^T\boldsymbol{x} \\ A\boldsymbol{x} &\leq \boldsymbol{b} \\ \boldsymbol{x} &\geq 0 \end{aligned}$$

Linear Programming (LP)

$$\begin{aligned} \max\ & \boldsymbol{c}^T\boldsymbol{x} \\ A\boldsymbol{x} &\leq \boldsymbol{b} \\ \boldsymbol{x} &\geq 0 \\ \boldsymbol{x} &\ \text{integer} \end{aligned}$$

Integer Linear Programming (ILP)

$$\begin{aligned} \max\ & \boldsymbol{c}^T\boldsymbol{x} \\ A\boldsymbol{x} &\leq \boldsymbol{b} \\ \boldsymbol{x} &\in \{0,1\}^n \end{aligned}$$

Binary Integer Program (BIP)
0/1 Integer Programming

$$\begin{aligned} \max\ & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{h}^T\boldsymbol{y} \\ A\boldsymbol{x} + G\boldsymbol{y} &\leq \boldsymbol{b} \\ \boldsymbol{x} &\geq 0 \\ \boldsymbol{y} &\geq 0 \\ \boldsymbol{y} &\ \text{integer} \end{aligned}$$

Mixed Integer Linear Programming (MILP)

$$\begin{aligned} \max\ & f(\boldsymbol{x}) \\ g(\boldsymbol{x}) &\leq \boldsymbol{b} \\ \boldsymbol{x} &\ \text{integer} \end{aligned}$$

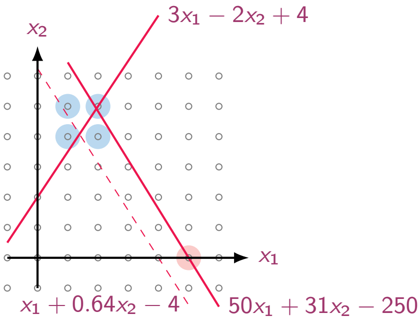
Integer Non-linear Programming (INLP)

Mathematical Programming: Modeling

- Find out exactly what the decision maker needs to know:
  - which investment?
  - which product mix?
  - which job j should a person i do?
- Define **Decision Variables** of suitable type (continuous, integer valued, binary) corresponding to the needs and **Known Parameters** corresponding to given data.
- Formulate **Objective Function** computing the benefit/cost
- Formulate mathematical **Constraints** indicating the interplay between the different variables.

Rounding

$$\begin{aligned} \max\ & 100x_1 + 64x_2 \\ & 50x_1 + 31x_2 \leq 250 \\ & 3x_1 - 2x_2 \geq -4 \\ & x_1, x_2 \in \mathbb{Z}_0^+ \end{aligned}$$



Note: rounding does not help in the example above!

LP optimum $(376/193, 950/193)$
IP optimum $(5, 0)$

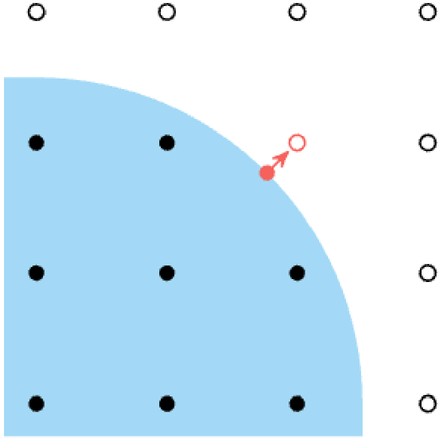
$\rightsquigarrow$ feasible region convex but not continuous: Now the optimum can be on the border (vertices) but also **internal**.

Possible way: solve the **relaxed** problem.

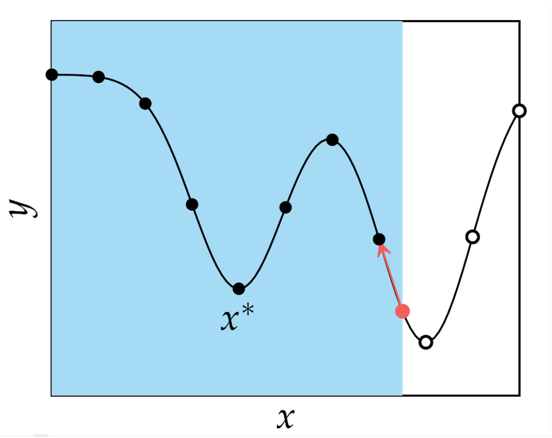
- If solution is **integer**, done.
- If solution is **rational** (never irrational) try rounding to the nearest integers (but may exit feasibility region)
  if in $\mathbb{R}^2$ then 2^2 possible roundings (up or down)
  if in $\mathbb{R}^n$ then 2^n possible roundings (up or down)

Rounding

Rounding to infeasible point



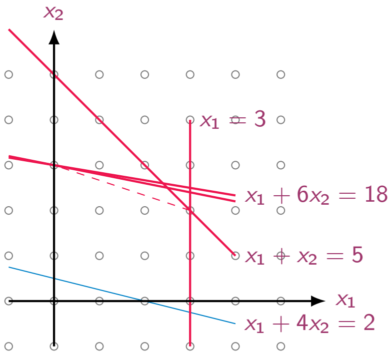
Rounding to suboptmal point



If A is integral, the error of a rounded solution can be bounded

Cutting Planes

$$
\begin{array}{rcrcl}
\max & x_1 & + & 4x_2 & \\
 & x_1 & + & 6x_2 & \leq 18 \\
 & x_1 & & & \leq 3 \\
 & & & x_1, x_2 & \geq 0 \\
 & & & x_1, x_2 & \text{integers}
\end{array}
$$



Cuts: Useful valid inequalities: they are satisfied by all integer solutions but not by all relaxed solutions

Cutting Plane Method

- The cutting plane method solves the relaxed LP, adds linear constraints, then repeats until the solution is exact
- Solves mixed integer programs exactly
- The linear constraints are chosen such that all discrete points are still feasible, but the relaxed solution is not

Chvatal-Gomory's Cutting Plane Algorithm

- Recall that we can partition a vertex (and also an optimal one $\boldsymbol{x}^*$) as

$$A_B \boldsymbol{x}_B^* + A_N \boldsymbol{x}_N^* = \boldsymbol{b}$$

- Using the method of Gomory's cut, we can add an additional inequality constraint for each nonintegral dimension

$$x_b^* - \lfloor x_b^* \rfloor - \sum_{j \in N} \left(\bar{A}_{bj} - \lfloor \bar{A}_{bj} \rfloor \right) x_j \leq 0 \qquad \bar{A} = A_B^{-1} A_N$$

- This "cuts out" the relaxed solution $\boldsymbol{x}^*$

$$\underbrace{x_b^* - \lfloor x_b^* \rfloor}_{>0} - \underbrace{\sum_{j \in N} \left(\bar{A}_{bj} - \lfloor \bar{A}_{bj} \rfloor \right) x_j^*}_{=0} > 0$$

Branch and Bound

- Algorithm for efficiently searching the very large set of solution possibilities (first proposed by Ailsa Land and Alison Doig, "An automatic method of solving discrete programming problems", 1960)

- **Branching** is dividing the domain into sections

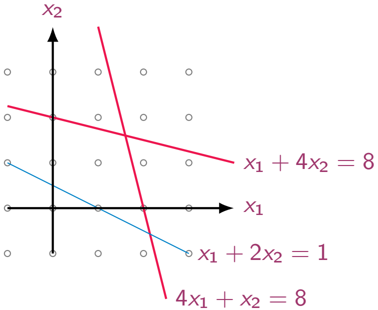
- **Bounding** is keeping track of the best solution so far and rejecting regions that cannot improve upon it

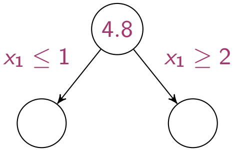
- In the worst case, the algorithm has to search all possibilities but in practice it works very well. Combined with cutting planes, this approach forms the basis of many commercial MIP solvers.

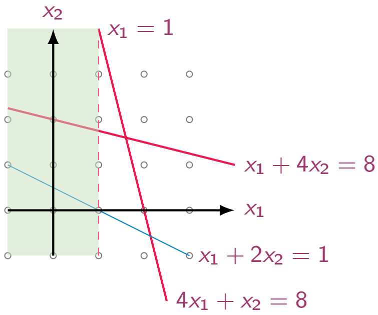
Branch and Bound

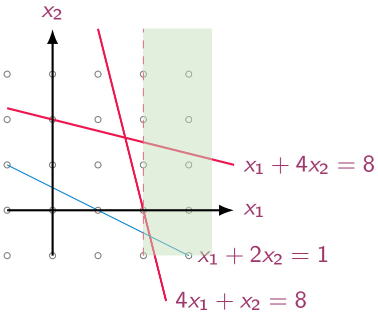
Example

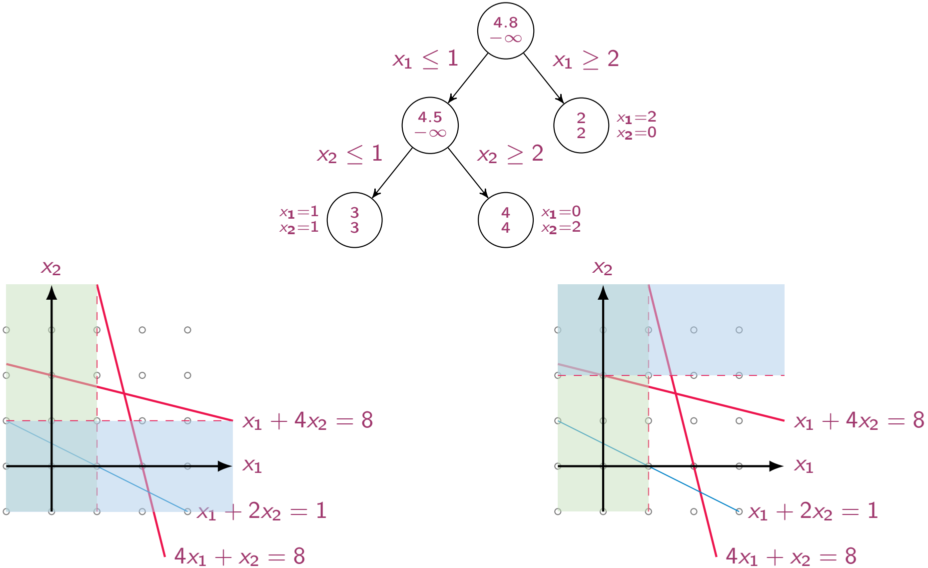
$$
\begin{aligned}
\max \quad x_1 &+ 2x_2 \\
x_1 &+ 4x_2 \leq 8 \\
4x_1 &+ x_2 \leq 8 \\
x_1, x_2 &\geq 0, \text{integer}
\end{aligned}
$$




4.8
$x_1 \leq 1$
$x_1 \geq 2$


x_2
$x_1 = 1$
$x_1 + 4x_2 = 8$
x_1
$x_1 + 2x_2 = 1$
$4x_1 + x_2 = 8$

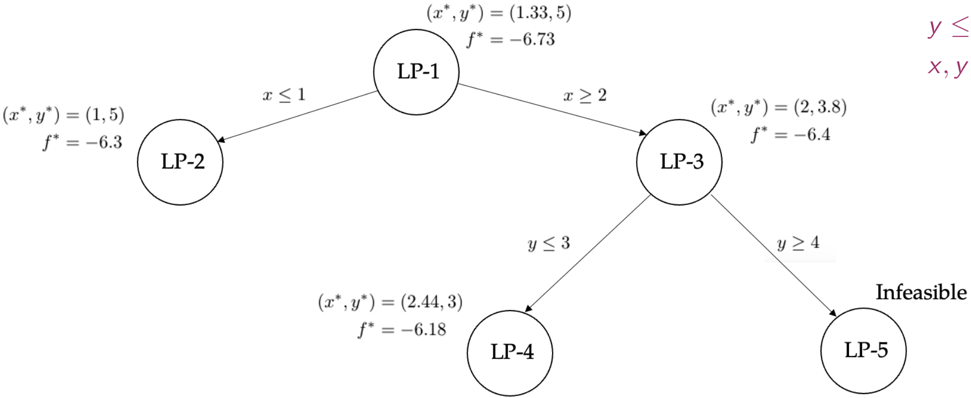

x_2
$x_1 + 4x_2 = 8$
x_1
$x_1 + 2x_2 = 1$
$4x_1 + x_2 = 8$


4.8
$-\infty$
$x_1 \leq 1$
$x_1 \geq 2$
4.5
$-\infty$
2
2
$x_1=2$
$x_2=0$
$x_2 \leq 1$
$x_2 \geq 2$
$x_1=1$
$x_2=1$
3
3
4
4
$x_1=0$
$x_2=2$
x_2
x_1
$x_1 + 4x_2 = 8$
$x_1 + 2x_2 = 1$
$4x_1 + x_2 = 8$
x_2
x_1
$x_1 + 4x_2 = 8$
$x_1 + 2x_2 = 1$
$4x_1 + x_2 = 8$

Branch and Bound: Pruning

$$\underset{x,y}{\text{minimize}} \; -1.3x - y$$

$$1.8x + y \leq 7.4$$
$$y \leq 5$$
$$x, y \in \mathbb{Z}$$



Pruning by: integrality, bounding, infeasibiliy.

Outline

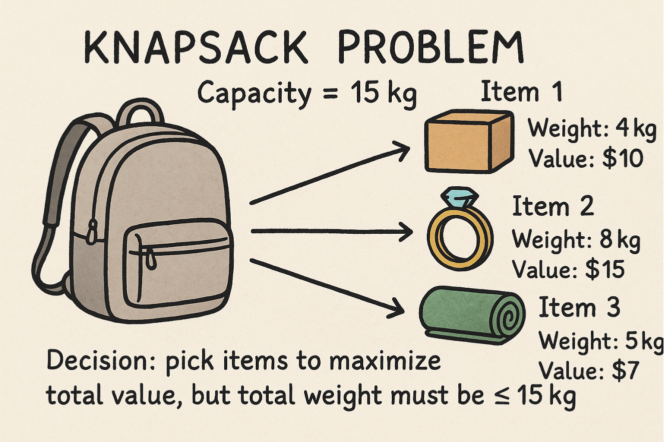
1. Integer Linear Programming

2. Dynamic Programming

Dynamic Programming

- Applied to problems with **optimal substructure** and **overlapping subproblems**
- Optimal substructure means an optimal solution can be constructed from optimal solutions to its subproblems
- Overlapping subproblems means solving each subproblem separately requires repeating certain operations
- **Dynamic programming** begins with desired problem and recurses down to smaller and smaller subproblems, retrieving the value of previously solved problems as necessary
- Principle of Optimality (known as Bellman Optimality Conditions): Suppose that the solution of a problem is the result of a sequence of n decisions $D_1, D_2, ..., D_n$; if a given sequence is optimal, then the first k decisions must be optimal, but also the last $n - k$ decisions must be optimal
- DP breaks down the problem into stages, at which decisions take place, and find a recurrence relation that relates each stage with the previous one

Example 1: Knapsack Problem



- Trying to pack some number of items into a backpack
- Limited space in the backpack
- Each item has a specified value and size
- What is the best subset of items to include?

A MILP Formulation

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & -\sum_{i=1}^{n} v_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\ & x_i \in \{0, 1\} \text{for } i = 1, 2, \ldots, n \end{aligned}$$

Dynamic Programming

- Let knapsack(i, w) be the maximum value achievable using the first i items and a knapsack capacity w.

- Consider the ith item. You can either use it or not.

- If you don't use it, then the value of your knapsack will be

$$\text{knapsack}(i-1, w)$$

- If you use it, then the value of your knapsack will be

$$\text{knapsack}(i-1, w-w_i) + v_i$$

The Recursion

- For each item i and weight w:

$$\text{knapsack}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \max \begin{cases} \text{knapsack}(i-1, w) & \text{(discard new item)} \\ \text{knapsack}(i-1, w - w_i) + v_i & \text{(include new item)} \end{cases} & \text{if } w_i \leq w \\ \text{knapsack}(i-1, w) & \text{if } w_i > w \end{cases}$$

- Optimal solution:

$$z^* = \text{knapsack}(n, W)$$

and trace back to find the items collected.

Example

Capacity = 5 kg

Item	Weight	Value
1	1 kg	$1
2	3 kg	$4
3	4 kg	$5

Step 1: Initialize
Create a table with (number of items + 1) rows and (capacity + 1) columns, filled with zeros.

```
  w = 0 1 2 3 4 5
i=0 | 0 0 0 0 0 0
i=1 | 0
i=2 | 0
i=3 | 0
```

Step 2: Fill it step by step

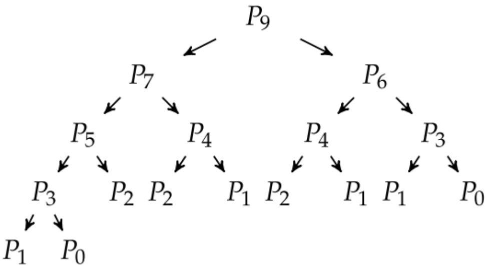
```
Item 1 (w_1=1, v_1=1):
  w = 0 1 2 3 4 5
i=0 | 0 0 0 0 0 0
i=1 | 0 1 1 1 1 1
i=2 | 0
i=3 | 0
```

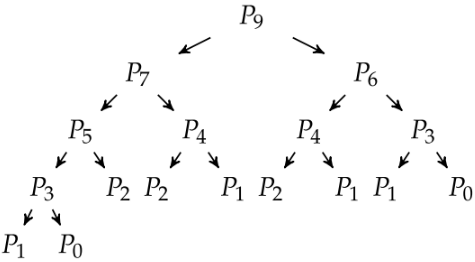
```
Item 2 (w_2=3, v_2=4):
  w = 0 1 2 3 4 5
i=0 | 0 0 0 0 0 0
i=1 | 0 1 1 1 1 1
i=2 | 0 1 1 4 5 5
i=3 | 0

Item 3 (w_3=4, v_3=5):
  w = 0 1 2 3 4 5
i=0 | 0 0 0 0 0 0
i=1 | 0 1 1 1 1 1
i=2 | 0 1 1 4 5 5
i=3 | 0 1 1 4 5 6
```

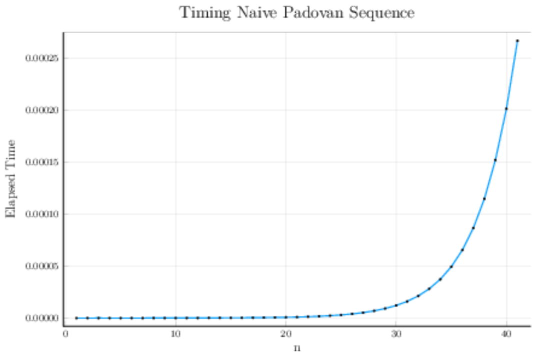
Example 2: Padovan Sequence

$$P_n = P_{n-2} + P_{n-3} \qquad\qquad P_0 = P_1 = P_2 = 1$$

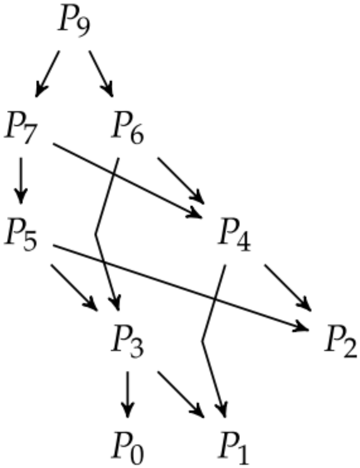
```python
def padovan_naive(n, P=None):
    if n < 3:
        return 1
    else:
        return padovan_naive(n - 2, P) + padovan_naive(n - 3, P)
```



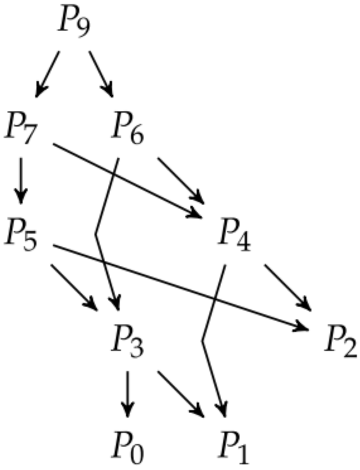

P_9
P_7
P_6
P_5
P_4
P_4
P_3
P_3
P_2
P_2
P_1
P_2
P_1
P_1
P_0
P_1
P_0

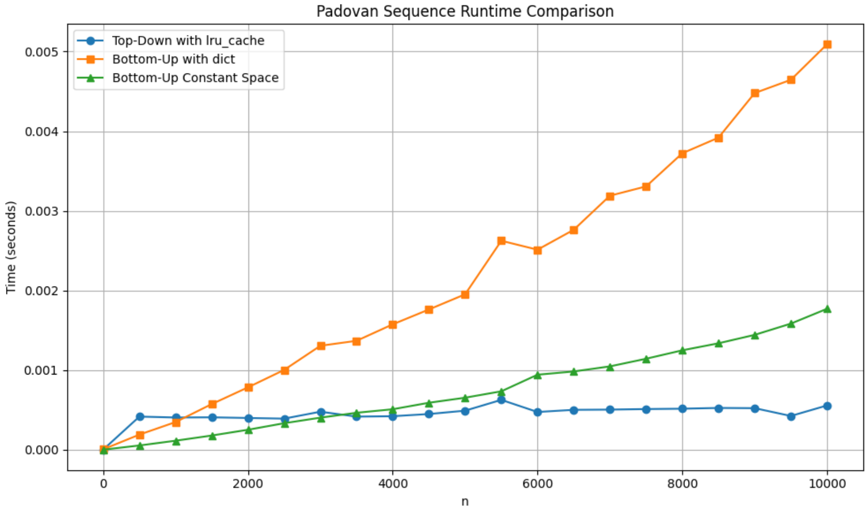

Timing Naive Padovan Sequence
Elapsed Time
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
0
10
20
30
40
n

```python
def padovan_topdown(n, P=None):
    if P is None:
        P = {}
    if n not in P:
        if n < 3:
            P[n] = 1
        else:
            P[n] = padovan_topdown(n - 2, P) + ↪
                ↪padovan_topdown(n - 3, P)
    return P[n]

def padovan_bottomup(n):
    P = {0: 1, 1: 1, 2: 1}
    for i in range(3, n + 1):
        P[i] = P[i - 2] + P[i - 3]
    return P[n]
```



```python
from functools import lru_cache

@lru_cache(maxsize=None)
def padovan_topdown(n):
    if n < 3:
        return 1
    return padovan_topdown(n - 2) + ↪
        ↪padovan_topdown(n - 3)

def padovan_bottomup_const(n):
    if n < 3:
        return 1
    p0, p1, p2 = 1, 1, 1  # Corresponds to P[0],↪
        ↪ P[1], P[2]
    for _ in range(3, n + 1):
        p_next = p0 + p1
        p0, p1, p2 = p1, p2, p_next
    return p2
```




Padovan Sequence Runtime Comparison
Top-Down with lru_cache
Bottom-Up with dict
Bottom-Up Constant Space
Time (seconds)
0.000
0.001
0.002
0.003
0.004
0.005
0
2000
4000
6000
8000
10000
n

Example 3: Traveling Salesman Problem

https://www.math.uwaterloo.ca/tsp/

Principle of Optimality

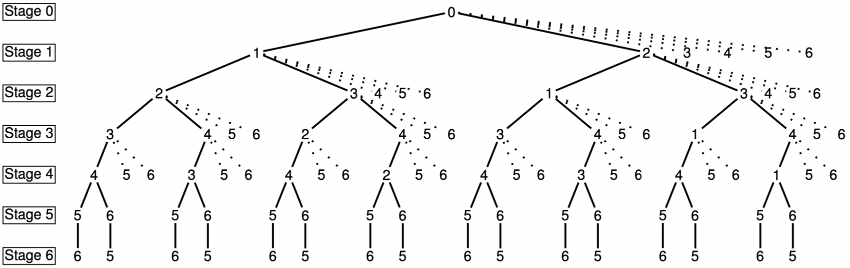
The TSP asks for the shortest tour that starts from 0, visits all cities of the set $C = \{1, 2, \ldots, n\}$ exactly once, and returns to 0, where the cost to travel from i to j is c_{ij} (with $(i, j) \in A$)
If the optimal solution of a TSP with six cities is $(0, 1, 3, 2, 4, 6, 5, 0)$, then...

- the optimal solution to visit $\{1, 2, 3, 4, 5, 6\}$ starting from 0 and ending at 5 is $(0, 1, 3, 2, 4, 6, 5)$
- the optimal solution to visit $\{1, 2, 3, 4, 6\}$ starting from 0 and ending at 6 is $(0, 1, 3, 2, 4, 6)$
- the optimal solution to visit $\{1, 2, 3, 4\}$ starting from 0 and ending at 4 is $(0, 1, 3, 2, 4)$
- the optimal solution to visit $\{1, 2, 3\}$ starting from 0 and ending at 2 is $(0, 1, 3, 2)$
- the optimal solution to visit $\{1, 3\}$ starting from 0 and ending at 3 is $(0, 1, 3)$
- the optimal solution to visit 1 starting from 0 is $(0, 1)$

$\rightsquigarrow$ The optimal solution is made up of a number of optimal solutions of smaller subproblems

Enumerate All Solutions of the TSP

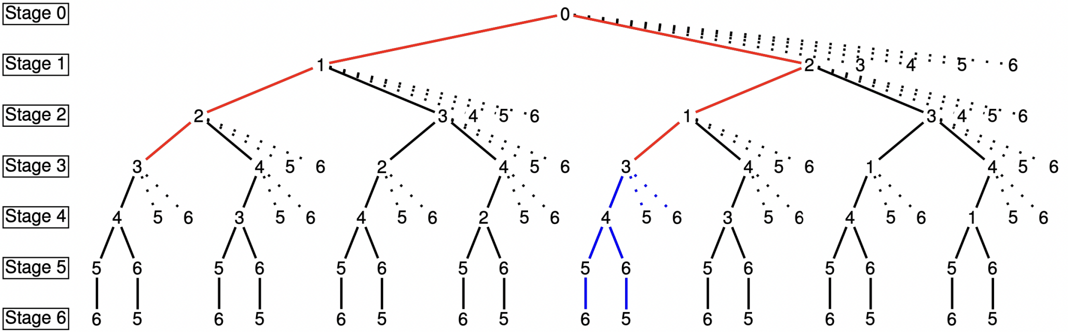
- A solution of a TSP with n cities derives from a sequence of n decisions, where the kth decision consists of choosing the kth city to visit in the tour



- The number of nodes (or states) grows exponentially with n
- At stage k, the number of states is $\binom{n}{k}k!$
- With $n = 6$, at stage $k = 6$, 720 states are necessary

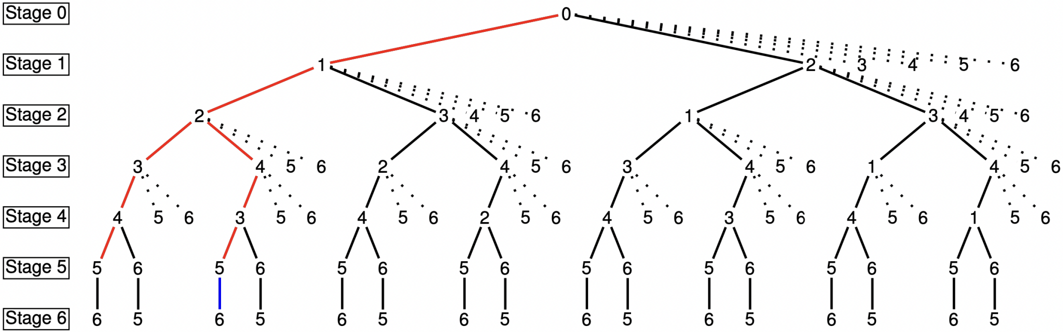
$\rightsquigarrow$ DP finds the optimal solution by implicitly enumerating all states but actually generating only some of them

Are All States Necessary?



If path $(0, 1, 2, 3)$ costs less than $(0, 2, 1, 3)$, the optimal solution cannot be found in the blue part of the tree

Are All States Necessary?



If path $(0, 1, 2, 3, 4, 5)$ costs less than $(0, 1, 2, 4, 3, 5)$, the optimal solution cannot be found in the blue part of the tree

Are All States Necessary?

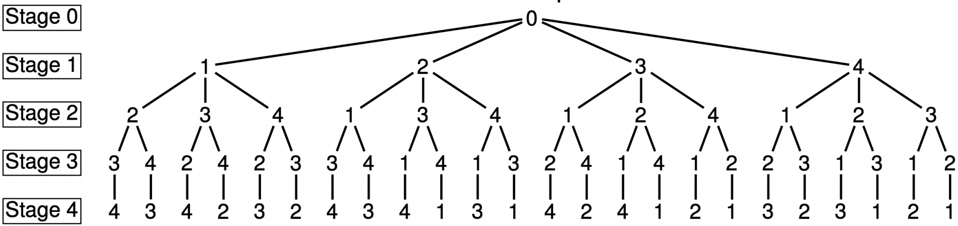
- At stage k $(1 \leq k \leq n)$, for each subset of cities $S \subseteq C$ of cardinality k, it is necessary to have only k states (one for each of the cities of the set S)
- At state $k = 3$, given the subset of cities $S = \{1, 2, 3\}$, three states are needed:
  - the shortest-path to visit S by starting from 0 and ending at 1
  - the shortest-path to visit S by starting from 0 and ending at 2
  - the shortest-path to visit S by starting from 0 and ending at 3
- At stage k, $\binom{n}{k}k$ states are required to compute the optimal solution (not $\binom{n}{k}k!$)

#States n = 6

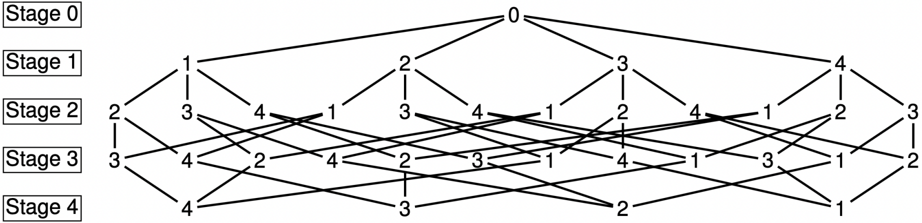
Stage	$\binom{n}{k}k!$	$\binom{n}{k}k$
1	6	6
2	30	30
3	120	60
4	360	60
5	720	30
6	720	6

Complete Trees with n=4

Enumeration of all paths



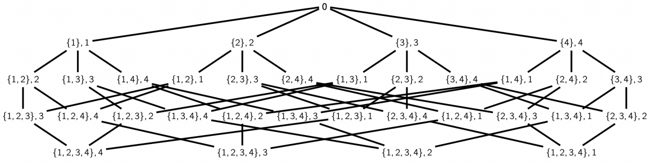
Implicit enumeration of all paths



Dynamic Programming Recursion for the TSP I

- Given a subset $S \subseteq C$ of cities and $k \in S$, let $f(S,k)$ be the optimal cost of starting from 0, visiting all cities in S, and ending at k
- Begin by finding $f(S,k)$ for $|S| = 1$, which is $f(\{k\},k) = c_{0k}, \forall k \in C$
- To compute $f(S,k)$ for $|S| > 1$, the best way to visit all cities of S by starting from 0 and ending at k is to consider all $j \in S \setminus \{k\}$ immediately before k, and look up $f(S \setminus \{k\}, j)$, namely

$$f(S,k) = \min_{j \in S \setminus \{k\}} \{f(S \setminus \{k\}, j) + c_{jk}\}$$



- The optimal solution cost z^* of the TSP is $z^* = \min_{k \in C}\{f(C,k) + c_{k0}\}$

Dynamic Programming Recursion for the TSP II

DP Recursion from [Held and Karp (1962)]

1. **Initialization**. Set $f(\{k\}, k) = c_{0k}$ for each $k \in C$

2. **RecursiveStep**. For each stage $r = 2, 3, ..., n$, compute

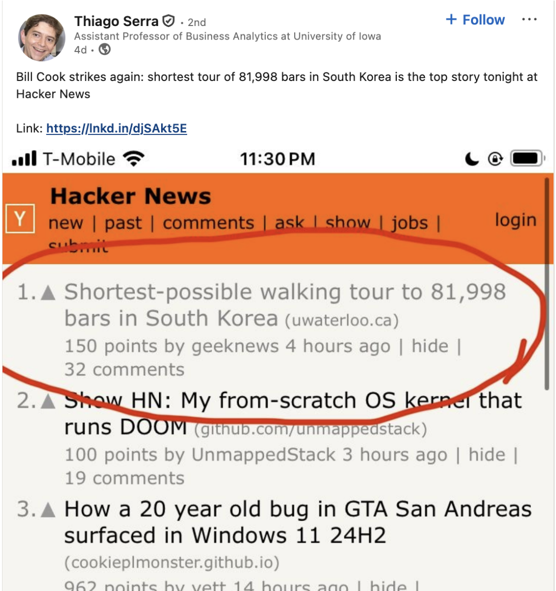
$$f(S, k) = \min_{j \in S \setminus \{k\}} \{f(S \setminus \{k\}, j) + c_{jk}\} \forall S \subseteq C : |S| = r \text{ and } \forall k \in S$$

3. **Optimal Solution**. Find the optimal solution cost z^* as

$$z^* = \min_{k \in C} \{f(C, k) + c_{k0}\}$$

- With the DP recursion, TSP instances with up to 25 - 30 customers can be solved to optimality; other solution techniques (i.e., branch-and-cut) are able to solve TSP instances with up to... 85900 customers
- Nonetheless, DP recursions represents the state-of-the-art solution techniques to solve a wide variety of PDPs

TSP


Thiago Serra · 2nd
+ Follow
Assistant Professor of Business Analytics at University of Iowa
4d ·
Bill Cook strikes again: shortest tour of 81,998 bars in South Korea is the top story tonight at Hacker News
Link: https://lnkd.in/djSAkt5E
T-Mobile
11:30 PM
Hacker News
Y
new | past | comments | ask | show | jobs | submit
login
1. Shortest-possible walking tour to 81,998 bars in South Korea (uwaterloo.ca)
150 points by geeknews 4 hours ago | hide | 32 comments
2. Show HN: My from-scratch OS kernel that runs DOOM (github.com/unmappedstack)
100 points by UnmappedStack 3 hours ago | hide | 19 comments
3. How a 20 year old bug in GTA San Andreas surfaced in Windows 11 24H2 (cookieplmonster.github.io)

Summary

- Discrete optimization problems require that the design variables be chosen from discrete sets.
- Relaxation, in which the continuous version of the discrete problem is solved, is by itself an unreliable technique for finding an optimal discrete solution but is central to more sophisticated algorithms.
- Many combinatorial optimization problems can be framed as an integer program, which is a linear program with integer constraints.
- Both the cutting plane and branch and bound methods can be used to solve integer programs efficiently and exactly. The branch and bound method is quite general and can be applied to a wide variety of discrete optimization problems.
- Dynamic programming is a powerful technique that exploits optimal overlapping substructure in some problems.