# Dynamic Programming

Richard de Neufville
Professor of Engineering Systems
and of
Civil and Environmental Engineering
MIT



# Objective

- To develop "dynamic programming", the method used in lattice valuation of flexibility
  - Its optimization procedure: "Implicit Enumeration"
  - Assumptions: Separability and Monotonicity
- To show its wide applicability
  - ➢ Analysis of Flexible Designs
  - Sequential problems: routing and logistics; inventory plans; replacement policies; reliability
  - Non-sequential problems: investments

## Outline

1. Why in this course?
2. Basic concept: Implicit Enumeration
   - Motivational Example
3. Key Assumptions
   - Independence (Separability) and Monotonicity
4. Mathematics
   - Recurrence Formulas
5. Example
6. Types of Problems DP can solve
7. Summary

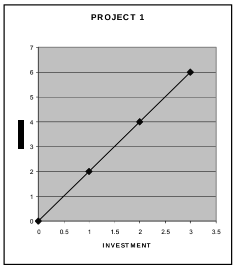


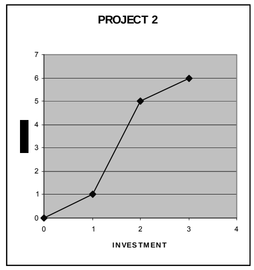
## Why in this course?

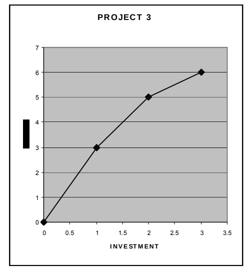
- DP is used to find optimum exercise of flexibility (options) in lattice that…
- generally has non-convex feasible region
  - Why is this?
  - Exponential growth ; also Flexibility to choose
- This presentation gives general method, so you understand it at deeper level
- DP used in lattice is simple version
  - only 2 states compared at any time

# Motivational Example

## Consider Possible Investments in 3 Projects



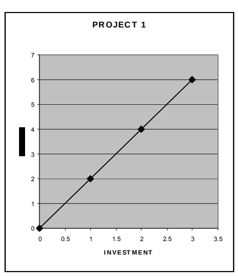



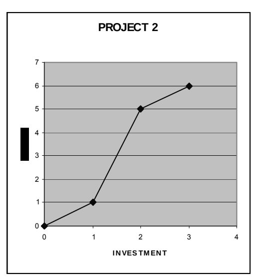

**What is best investment of 1st unit? P3 ➔ +3**

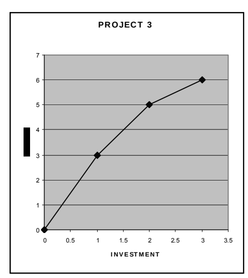
**Of 2nd? 3rd? P1 or P3 ➔ +2, +2 Total = 7**



# Motivational Example: Best Solution




**Optimum Allocation is Actually (0, 2, 1) ➔ 8**

**Marginal Analysis misses this….**
**because Feasible Region is not convex**

## Point of Example

- Non-convex feasible region "hides" optimum
- Marginal analysis ,"hill climbing" methods (such as linear programming) to search for optimum not appropriate in these cases
- Not appropriate for lattice models in particular

- We need to search entire space of possibilities
- This is what "Dynamic Programming" does to define optimum solution



## Semantic Note

- "Dynamic" Programming so named because
  - Originally associated with movements through time and space (e.g., aircraft gaining altitude, thus "dynamic")
  - "programming" by analogy to "linear programming" and other forms of optimization
- Approach useful in many cases that are not "dynamic" – such as motivational example
- Lattice model is "dynamic" as it models evolution across time

## Basic Solution Strategy

- Enumeration is basic concept
  - This means evaluating “all” the possibilities
  - Checking “all” possibilities, we must find best
- No assumptions about regularity of Objective Function
- Means that DP can optimize over
  - Non-Convex Feasible Regions
  - Discontinuous, Integer Functions
  - Which other optimization techniques cannot do
- HOWEVER…



## Curse of Dimensionality

- Number of Possible Designs very large
- Example: a simple development of 2 sites, for 4 sizes of operations over 3 periods
  - Number of Combinations in 1 period = $4^2$ = 16
  - Possibilities over 3 periods = $16^3$ = 4096
- General size of design space is exponential = $[\ (\text{Size})^{\text{locations}}]^{\text{periods}}$
  - Actual enumeration is impractical
- In lattice model…. See next slide

## The Curse -- in lattice model

**OUTCOME LATTICE**

| | | | | | | |
|---|---|---|---|---|---|---|
| 100.00 | 120.00 | 144.00 | 172.80 | 207.36 | 248.83 | 298.60 |
| | 80.00 | 96.00 | 115.20 | 138.24 | 165.89 | 199.07 |
| | | 64.00 | 76.80 | 92.16 | 110.59 | 132.71 |
| | | | 51.20 | 61.44 | 73.73 | 88.47 |
| | | | | 40.96 | 49.15 | 58.98 |
| | | | | | 32.77 | 39.32 |
| | | | | | | 26.21 |

- **End states = N**
- **Total States ~ Order of only $N^2/2$**
- **Number of paths ~ Order of $2^N$…**
  - **To reach each state at last stage = 1 + 6 +13 +16 + 13 + 6 +1 = 46 paths**



## Concept of Implicit Enumeration

- **Complete Enumeration Impractical**
- **We use "implicit enumeration" (IE)**
- **IE considers all possibilities in principle**
- **without actually doing so (thus "implicit")**
- **Exploits features of problem to**
  - **Identify groups of possibilities that are "<u>dominated</u>" – sets that all demonstrably inferior**
  - **Reject these groups -- not just single possibilities**
  - **Vastly reduce dimensionality of enumeration**

## Effect of Implicit Enumeration

- Because IE can reject groups of inferior (dominated) possibilities
- … it does not have to examine all
- … and reduces size of problem
- Specifically:  Size of numeration for DP
  - Order of (Size) (Locations)(Periods)
  - Multiplicative size, not exponential
  - This analysis computationally practical
- Examples illustrate what this means

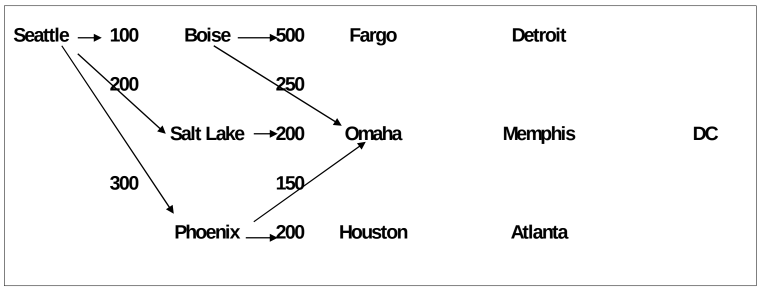


## Demonstration of IE

- Select a "dynamic" problem – logistic movement from Seattle to Washington DC
- Suppose that
  - there are 4 days to take trip…
  - Can go through several cities
  - There is a cost for the movement between any city  and possible city in next stage
- What is the minimum cost route?

## Possible routes through a node

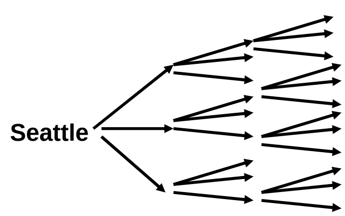
- Many routes, with link costs as in diagram
- Consider Omaha
  - 3 routes to get there, as shown
  - 3 routes from there => 9 routes via Omaha






## Notice that problem is a decision tree

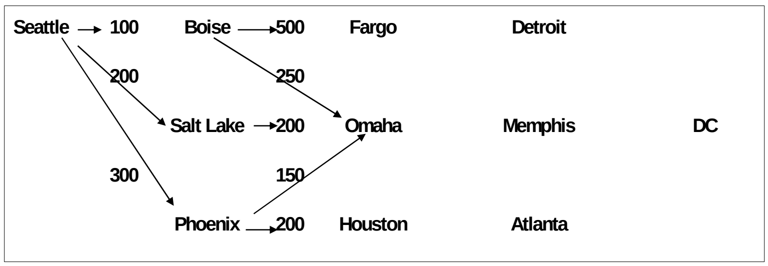
- In first stage, 3 choices
  - Another 3 in second, another 3 in third
  - In all 27 different paths
  - Same as a complicated decision tree




Not all
branches
drawn
for 3rd stage

## Instead of Costing all Routes… IE

- We find best cost to Omaha (350 via Boise)
- Salt Lake (400), Phoenix (450) routes dominated, "pruned" – we drop routes with those segments
- Thus don't look at all Seattle to DC routes



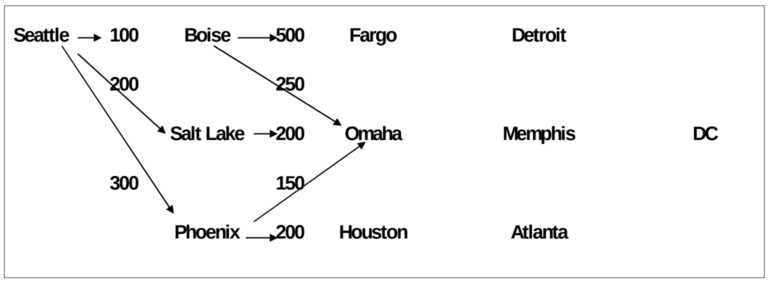



## Logic of "pruning' – dropping of routes

- There are many Seattle-DC routes that go through Omaha (9)
- A set of them (3) are between Seattle and Omaha – only one minimum cost level
  - (usually only 1 route, but more might be equal)
- The Seattle-Omaha routes (2) that are not minimum cost are "dominated"
- The routes that contain dominated sections (2 x 3 = 6) can be dropped from consideration

## Result: Fewer Combinations

- **Total Routes:**
  **= 3 to Omaha + 3 after = 6 = 3 x 2, not 9 = $3^2$**
- **Savings not dramatic here, illustrate idea**






## Dynamic Programming Definitions (1)

- **The Objective Function is G(X)**
- **Where X = ($X_1$,…. $X_N$ ) is a vector, the set of states at each of N "stages"**
  - **You can think of X as a "path" through problem**
- **To find the optimum "policy" or design, we have to find the $X_J$ that optimize G(X)**
  - **The object is to find the set of links that constitute the optimum overall route through problem**

## Concept of "Stages"

- A "Stage" is a transition through problem
  - From Seattle to first stop on the trip, for example
- "Stages" may
  - have a physical meaning, as in example,
  - or be conceptual (as the investments in later example) where a "stage" represents the next project or element or "knob" for system that we address



## Dynamic Programming Definitions (2)

- In parallel with G($\underline{X}$) which gives overall value
- $g_iX_J$ are the "return functions"
  - They define effect of being in $X_J$ state at the $i^{th}$ stage
- $g_iX_J$ denotes the functional form
  - Such as the costs of going for each link in a stage
- $X_J$ the different states at the $i^{th}$ stage
  - Such as being in Fargo, Omaha or Houston
- So that $G(\underline{X}) = [g_1X_1, \ldots.\ g_iX_J \ldots. g_MX_N]$

## Concept of "State"

- A "state" is one of possible locations, levels, or outcome for a stage
  - As a location: Fargo, Omaha or Houston
  - As a level: the amount invested (see later example)
- Each $g_iX_i$ is associated with a "stage"
  - Example: 1st Stage is from Seattle to Boise, etc
  - Thus $g_1X_J$ are costs from Seattle to Boise, etc
- … and with a "state" for each stage
  - It is the schedule of costs for stage 1, 2, etc…

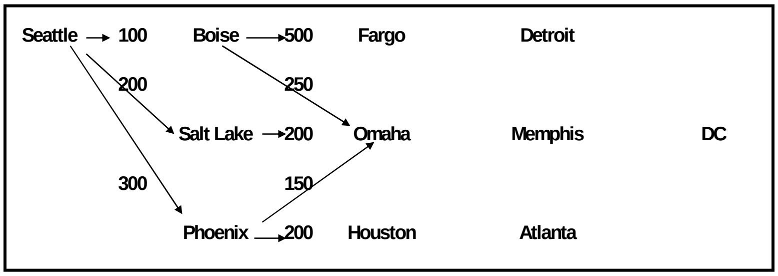


## Examples of States

- For cross-country shipment, there are 3 states (of system, not as "states" of USA) for 1st stage, Boise, Salt Lake and Phoenix
- For plane accelerating to altitude, a state might be defined by (speed, altitude) vector
- For investments, states might be $ invested
- If stage is "knob" we manipulate on system, "state" is the setting of the knob

## Stages and States

- "Stages" are associated with each move along trip
- Stage 1 consists of set of endpoints Boise, Salt Lake and Phoenix, Stage 2 the set of Fargo, Omaha and Houston; etc.
- "States" are possibilities in each Stage: Boise, Salt Lake, etc...






## Solution depends on Decomposition

- Must be able to "decompose" objective function G(X) into functions of individual "stages" $X_i$:

  $G(\underline{X}) = [g_1X_1, \ldots. g_MX_N]$

  - Example: cost of Seattle-DC trip can be decomposed into cost of 4 segments of which Seattle to Boise, Salt Lake or Phoenix is first
- This is the feature that permits us to
  - consider stages 1 by 1,
  - and thus to prune many logical possibilities

## Assumptions Needed

- **Necessary conditions for decomposition:**

  **Separability**

  **Monotonicity**

- **Another condition needed for DP:**

  **No Cyclic movement**
  **(always "forward")**



## Separability

- **Objective Function is separable if all $g_i X_i$ are independent of $g_J X_J$ for all J not equal to I**
- **In example, it is reasonable to assume that the cost of driving between any pair of cities is not affected by that between another pair**
- **However, not always so…**

## Monotonicity

- Objective Function is monotonic if: improvements in each $g_iX_i$ lead to improvements in Objective Function, that is if
- given $G(\underline{X}) = [g_iX_i , G'(\underline{X'}) ]$ where $\underline{X} = [X_i , \underline{X}']$
- for all $g_iX'_i \geq g_iX''_i$ where $X'_i , X''_i$ different $X_i$
- It is true that $[g_iX'_i , G'(\underline{X}) ] \geq [g_iX''_i , G'(\underline{X}) ]$
- For example…



## When are functions Monotonic?

- Additive functions always monotonic
- Multiplicative functions monotonic only

  if $g_iX_i$ are <u>non-negative</u>, real

## Solution Strategy

- **Two Steps**
  - Partial optimization at each stage
  - Repetition of process for all stages
- **This is the process used to value flexibility (options) through the lattice**
  - At each stage (period), for each state (possible outcome for system)
  - Process chooses better of using flexibility (exercising option) -- or not using it



## Cumulative Return Function

- **Result of Optimization at each stage and state is the "cumulative return function" = $f_S$ (K)**
- **$f_S$ (K) denotes best value for being in state K, having passed through previous S stages**
- **Example: $f_2$ (Omaha) = 350**
- **Defined in terms of best over previous stages and return functions for this stage, $g_iX_J$ :**

$$f_S (K) = \text{Max or Min of } [g_iX_J , f_{S-1} (K) ]$$

**(note: K understood to be a variable)**

## Mathematics: Recurrence formulas

- Transition from one stage to next is via a "recurrence formula"
- or equivalent analysis (see lattice valuation)
- Formally, we seek the best we can obtain to any specified level K, by finding the best combination of possible $g_i X_J$ and $f_{S-1}$ (K)

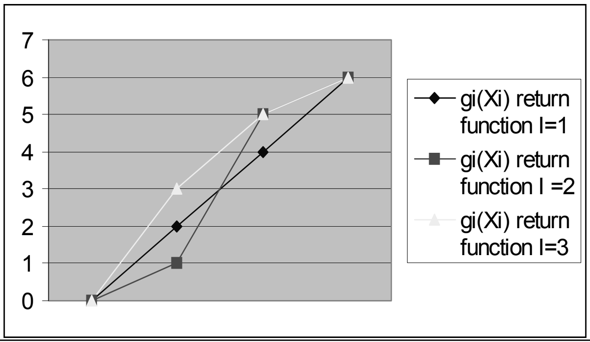


## Application of Recurrence formulas

- For Example: Consider the Maximization investments in independent projects
  - Each project is a "stage"
  - Amount of Investment in each is its "state"
  - Objective Function Is Additive:
    Value = $\Sigma$ (value each project)
  - Recurrence formula: $f_i (K) = Max[g_i X_J + f_{i-1} (K- X_J)]$
  - that is: optimum for investing K over "i" stages
  = maximum of all combinations of investing level $X_J$ in stage "i" and (K- $X_J$ ) in previous stages

## Application to Investment Example

- 3 Projects, 4 Investment levels (0, 1, 2, 3)
- Objective: Maximum for investing 3 units
- Stages = projects ; States = investment levels






## Dynamic Programming Analysis (1)

- At 1st stage the cumulative return function identically equals return for $X_1$
- That is, $f_1(X_1)$, the best way to allocate resource over only one stage $\equiv g_1X_1$
  - There is no other choice
- So $f_1(0) = 0$
- $f_1(1) = 2$ ; $f_1(2) = 4$ ; $f_1(3) = 6$

## Dynamic Programming Analysis (2)

- **At 2nd stage, best way to spend:**

  **0 : is 0 on both 1st and 2nd stage (= 0) = $f_2$ (0)**

  **1 : either: 0 on 1st and 1 on 2nd stage (= 1)**
  **or: 1 on 1st and 0 on 2nd stage (= 2) BEST = $f_2$ (1)**

  **2 : 2 on 1st , and 0 on 2nd stage (= 4)**

  **1 on 1st, and 1 on 2nd stage (= 3)**

  **0 on 1st, and 2 on 2nd stage (= 5) BEST = $f_2$ (2)**

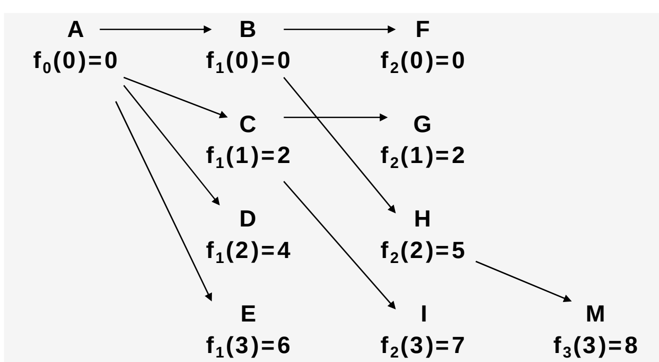
  **3: 4 Choices, Best allocation is (1,2) ➔ 7 = $f_2$ (3)**

  **These results, and the corresponding allocations, shown on next figures…**



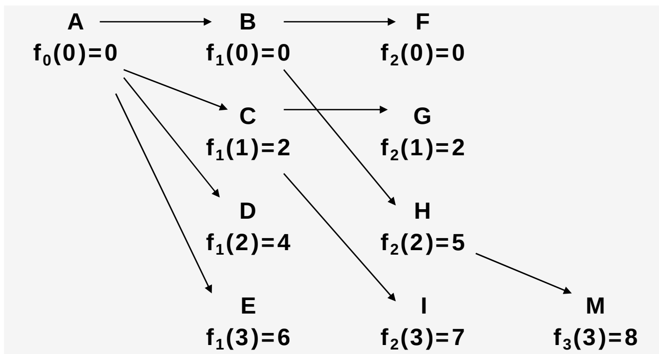
## Dynamic Programming Analysis (3)

- **LH Column: 0 in no Project $f_0(0) = 0$**
- **2nd Column: 0…3 in 1st project, e.g.: $f_1(2) = 4$**



## Dynamic Programming Analysis (4)

- **For 3rd stage (all 3 projects) we want optimum allocation of all 3 units: ➔ (0,2,1) ➔ $f_3(3) = 8$**






## Contrast DP and Marginal Analysis

- **Marginal Analysis:**
  - **reduces calculation burden by only looking at "best" slopes towards goal, discards others**
  - **Misses opportunities to take losses for later gains**
  - **approach ➔ 7**
- **Dynamic Programming :**
  - **Looks at "all" possible positions**
  - **But cuts out combinations that are dominated**
  - **Using independence return functions (value from a state does not depend on what happened before)**

## Classes of Problems suitable for DP

- Sequential, “Dynamic” Problems
  -- aircraft flight paths to maximize speed, altitude
  -- movement across territory (example used)
- Schedule, Inventory (Management over time)
- Reliability -- Multiplicative example, see text
- ➢ Flexibility (options) analysis!

- Non-Sequential: Investment Maximizations
  - Nothing Dynamic. Key is separability of projects



## Formulation Issues

- No standard (“canonical”) form
- Careful formulations required (see text)
- DP assumes discrete states
  - thus easily handles integers, discontinuity
  - in practice does not handle continuous variables
- DP handles constraints in formulation
  - Thus certain paths not defined or allowed
- Sensitivity analysis is not automatic

## Dynamic Programming Summary

- The method used to deal with lattices
- Solution by implicit enumeration
- Approach requires
  - separability, monotonicity -- and no cycles
- Careful formulation needed
- Useful for wide range of issues

-- particularly flexibility, options analyses!