Discrete Optimization

MA2827

Fondements de l'optimisation discrète

Dynamic programming, Branch-and-bound

https://project.inria.fr/2015ma2827/

Material based on the lectures of Erik Demaine at MIT and Pascal Van Hentenryck at Coursera

Outline

- Dynamic programming
  - Fibonacci numbers
  - Shortest paths
  - Text justification
  - Parenthesization
  - Edit distance
  - Knapsack
- Branch-and-bound
- More dynamic programming
  - Guitar fingering, Hardwood floor (Parquet)
  - Tetris, Blackjack, Super Mario Bros.

Dynamic programming

- Powerful technique to design algorithms
- Searches over exponential number of possibilities in polynomial time
- DP is a “controlled brute force”
- DP = recursion + reuse

Fibonacci numbers

Recurrence: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$

Task: compute F_n

Fibonacci numbers: naïve algorithm

Recurrence: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$
Task: compute F_n

Naïve algorithm:

```
def fib(n):
    if n <= 2:
        f =1
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

Fibonacci numbers: naïve algorithm

Recurrence: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$
Task: compute F_n

Naïve algorithm has exponential running time:
$T(n) = T(n-1) + T(n-2) + O(1) \geq F_n \approx \varphi^n$

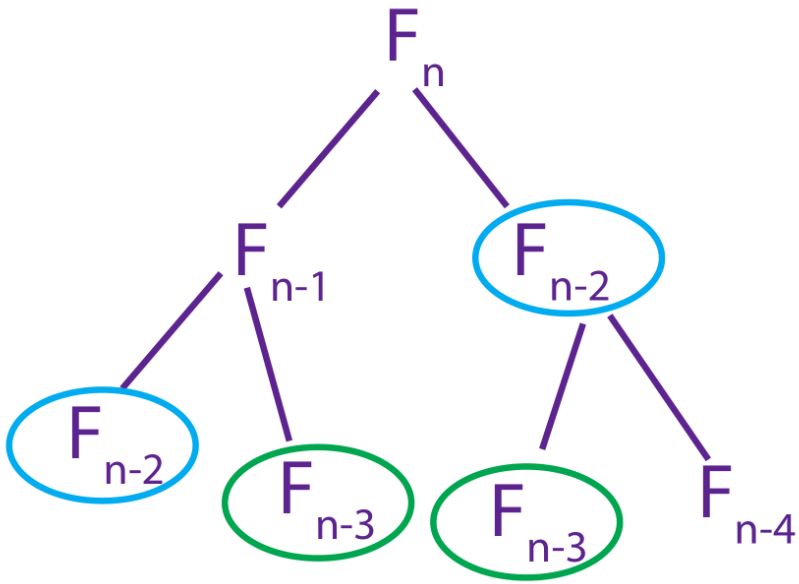
Easier:
$T(n) >= 2T(n-2) \quad => \quad T(n) = O(2^{0.5n})$

Fibonacci numbers: naïve algorithm

Recurrence: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$

Task: compute F_n

Naïve algorithm has exponential running time:



General idea: memoization

Recurrence: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$

Task: compute F_n

Memoized DP algorithm:

```
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n <= 2:  f =1
    else: f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```

General idea: memoization

Recurrence: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$
Task: compute F_n

Why memoized DP algorithm is efficient?

- fib(k) recurses only when called first time
- n nonmemoized calls for k=n, n-1, …, 1
- memoized calls take O(1) time
- overall running time is O(n)

Footnote: better algorithm is O(log n)

General idea: memoization

DP ≈ recursion + memoization

Memoize (save) & re-use solutions to subproblems

Time = #subproblems · time/subproblem

Memoized calls take constant time

General idea: bottom-up DP

Recurrence: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$

Task: compute F_n

Bottom-up DP algorithm:

```
def fib(n):
    memo = {}
    for k in range(1, n+1):
        if k <= 2: f =1
        else: f = memo[k-1] + memo[k-2]
        memo[k] = f
    return memo[n]
```

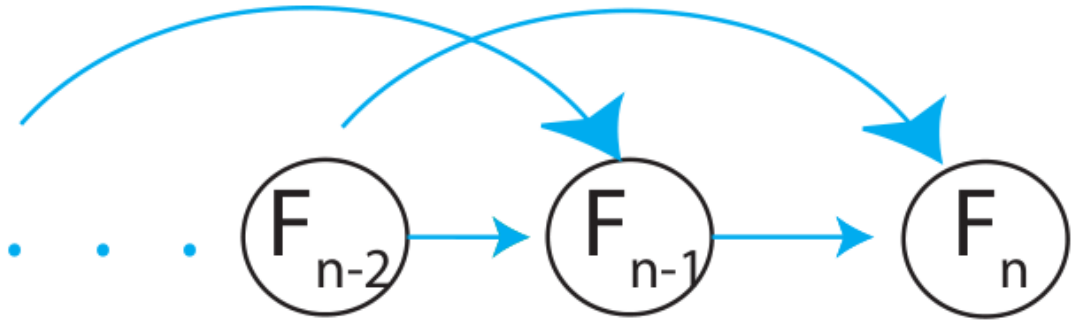
General idea: bottom-up DP

Recurrence: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$

Task: compute F_n

Bottom-up DP algorithm:

- Exactly the same computation as memoized DP
- Need topological sorting of the subproblems
- Dependency graph:



General idea: bottom-up DP

Recurrence: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$

Task: compute F_n

Bottom-up DP algorithm:

- Exactly the same computation as memoized DP
- Need topological sorting of the subproblems
- In practice is faster (no recursion calls)
- Analysis is easier
- Can save space

Shortest paths

Task: find shortest path $\delta(s, v)$ from s to v

Recurrence:

Shortest path = shortest path + last edge

$\delta(s, v) = \delta(s, u) + w(u, v)$

How do we know what is the last edge?

Guess!

Try all the guesses and pick the best

Shortest paths

Task: find shortest path $\delta(s, v)$ from s to v

Recurrence:

$$\delta(s, v) = \min\{\ \delta(s, u) + w(u, v) \mid (u, v) \in E\ \}$$

Memoized DP algorithm:

```
memo = {}
memo[s] = 0

def δ(s, v):
    if not v in memo:
        memo[v] = min{ ∞, δ(s, u) + w(u, v) | (u, v) ∈ E }
    return memo[v]
```

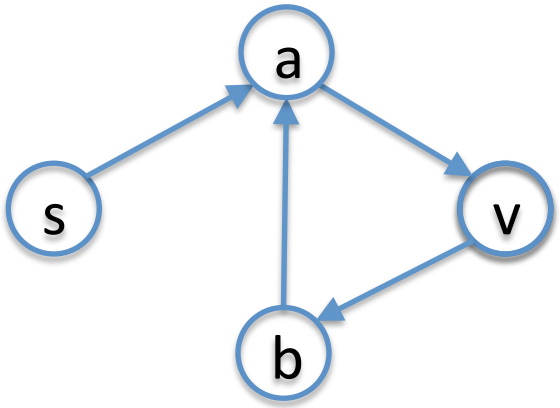
Shortest paths

Task: find shortest path $\delta(s, v)$ from s to v

Recurrence:

$$\delta(s, v) = \min\{ \ \delta(s, u) + w(u, v) \mid (u, v) \in E \ \}$$

Does this work?



Execution order:

1. $\delta(s, v)$
2. $\delta(s, a)$
3. $\delta(s, s)$
4. $\delta(s, b)$
5. $\delta(s, v)$ - infinite recursion

Shortest paths

Task: find shortest path $\delta(s, v)$ from s to v

Recurrence:

$$\delta(s, v) = \min\{ \ \delta(s, u) + w(u, v) \mid (u, v) \in E \ \}$$

Does this ever work?

- If there are no oriented loops (graph is a DAG) works fine
- Complexity $O(V + E)$

Shortest paths

Task: find shortest path $\delta(s, v)$ from s to v

Recurrence:

$$\delta(s, v) = \min\{ \ \delta(s, u) + w(u, v) \mid (u, v) \in E \ \}$$

Graph of subproblems has to be acyclic (DAG)!

(need this property to topologically sort)

Shortest paths

Task: find shortest path $\delta(s, v)$ from s to v

Recurrence:

$\delta(s, v) = \min\{ \ \delta(s, u) + w(u, v) \mid (u, v) \in E \ \}$

How do we fix the recursion?

Add more subproblems!

$\delta_k(s, v)$ = shortest path from s to v using at most k edges

Shortest paths

Task: find shortest path $\delta(s, v)$ from s to v

Recurrence:

$$\delta_k(s, v) = \min\{ \ \delta_{k-1}(s, u) + w(u, v) \mid (u, v) \in E \ \}$$

Base case:

$\delta_k(s, s) = 0, \ k \geq 0$

$\delta_0(s, u) = \infty, \ u \neq s$

Goal: $\delta_{V-1}(s, v)$

Time= #subproblem · time/subproblem

#subproblem = $O(V^2)$

time/subproblem = O(indegree of v)

Time = O(V E)

Bellman-Ford algorithm is dynamic programming!

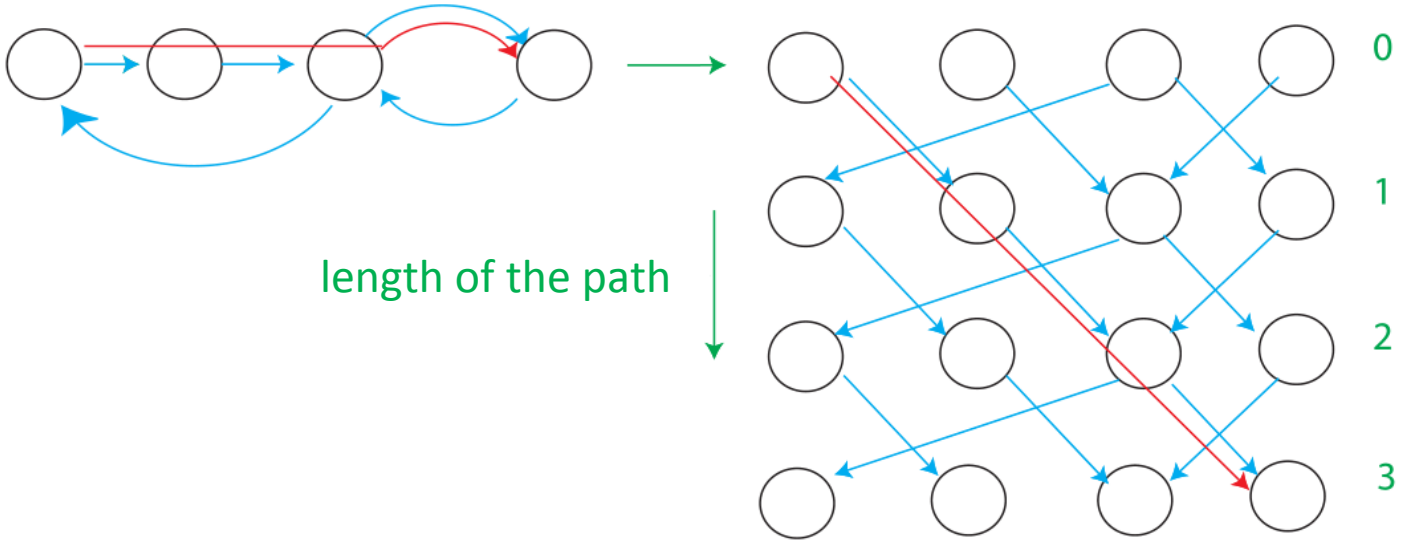
Shortest paths

Task: find shortest path $\delta(s, v)$ from s to v

Recurrence:

$$\delta_k(s, v) = \min\{ \ \delta_{k-1}(s, u) + w(u, v) \mid (u, v) \in E \ \}$$

Graph of subproblems:



Summary

- DP ≈ “careful brute force”
- DP ≈ recursion + memoization + guessing
- Divide the problem into subproblems that are connected to the original problem
- Graph of subproblems has to be acyclic (DAG)
- Time = #subproblems · time/subproblem

5 easy steps of DP

Analysis:

1. Define subproblems — #subproblems
2. Guess part of solution — #choices
3. Relate subproblems (recursion) — time/subproblem
4. Recurse + memoize — time
   OR build DP table bottom-up
   - check subprobs be acyclic / topological order
5. Solve original problem — extra time

5 easy steps of DP

	Fibonacci	Shortest paths
1. Subproblems	F_k, $1 \le k \le n$	$\delta_k(s, v)$, $v \in V$, $0 \le k \le V$
#subproblems	n	V^2
2. Guessing	F_{n-1}, F_{n-2}	edges coming into v
#choices	1	indegree(v)
3. Recurrence	$F_n = F_{n-1} + F_{n-2}$	$\delta_k(s, v) = \min\{ \delta_{k-1}(s, u) + w(u, v) \mid (u, v) \in E \}$
time/subproblem	O(1)	O(indegree(v))
4. Topological order	for k = 1, …, n	for k = 0, 1, …, V - 1 for v ∈ V
total time	O(n)	O(V E)
5. Original problem	F_n	$\delta_{V-1}(s, v)$
extra time	O(1)	O(1)

5 easy steps of DP

	Fibonacci	Shortest paths
1. Subproblems	F_k, $1 \le k \le n$	$\delta_k(s, v)$, $v \in V$, $0 \le k \le V$
#subproblems	n	V^2
2. Guessing	F_{n-1}, F_{n-2}	edges coming into v
#choices	1	indegree(v)
3. Recurrence	$F_n = F_{n-1} + F_{n-2}$	$\delta_k(s, v) = \min\{ \delta_{k-1}(s, u) + w(u, v) \mid (u, v) \in E \}$
time/subproblem	O(1)	O(indegree(v))
4. Topological order	for k = 1, …, n	for k = 0, 1, …, V - 1 for v ∈ V
total time	O(n)	O(V E)
5. Original problem	F_n	$\delta_{V-1}(s, v)$
extra time	O(1)	O(1)

Text justification

Task: split the text into “good” lines

Greedy algorithm (MS Word/Open Office):
put as many words on the first line, repeat

Optimal – dynamic programming! (LATEX)

Text justification

Task: split the text into “good” lines

Define badness(i, j) for line of words[i : j] from i, i+1, …, j-1

In LATEX:

- badness = ∞ if total length > page width
- badness = (page width – total length)^3

Goal: minimize overall badness

Text justification

Task: split the text into “good” lines
Goal: minimize overall badness

Subproblems:
min. badness for suffix words[i :]
#subproblems = O(n) where n = #words

Guesses:
what is the first word of the next line, j
#choices = n – i = O(n)

Recurrence:
DP(i) = min{ badness(i, j) + DP(j) | for j = i+1, …, n }
time/subproblem = O(n)

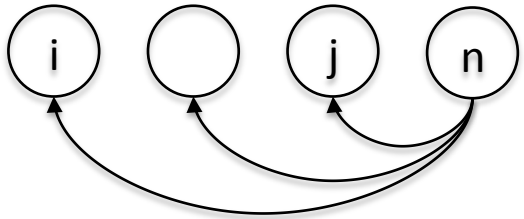
Text justification

Task: split the text into “good” lines

Goal: minimize overall badness

Topological order:

for i = n – 1, n – 2, …, 0



total time = $O(n^2)$

Final problem:

DP(0)

General idea: parent pointers

Task: split the text into “good” lines
Goal: minimize overall badness

How to find the actual justification?

Parent pointers – simple general idea

Remember which guess was best and save argmin

At the end, follow the parent pointers starting from the final problem

Useful subproblems for sequences

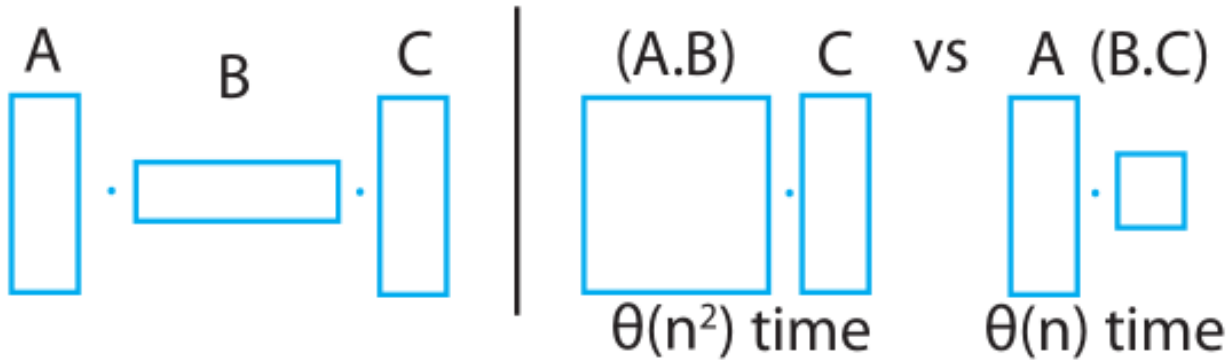
Input is a sequence/string x

Subproblems:

1. Suffixes x[i :]
   #subproblems = O(n)
2. Prefixes x[: i]
   #subproblems = O(n)
3. Substrings x[i : j]
   #subproblems = O(n^2)

Parenthesization

Task: Optimal evaluation of associative expression $A[0] \cdot A[1] \cdots A[n-1]$ (e.g., multiplying matrices)



Parenthesization

Task: Optimal evaluation of associative expression $A[0] \cdot A[1] \cdots A[n-1]$ (e.g., multiplying matrices)

Guesses:

outermost multiplication $\underbrace{(\cdots)}_{\uparrow k-1}\underbrace{(\cdots)}_{\uparrow k}$

#choices = O(n)

Subproblems:

Prefixes & suffixes?

NO

Recurrence:

Parenthesization

Task: Optimal evaluation of associative expression $A[0] \cdot A[1] \cdots A[n-1]$ (e.g., multiplying matrices)

Guesses:
outermost multiplication

#choices = O(n)

Subproblems:
Cost of substring A[i : j], from i to j-1
#subproblems = $O(n^2)$

Recurrence:
DP(i, j) = min{ DP(i, k) + DP(k, j) + cost of (A[i]…A[k-1]) (A[k]…A[j-1])
| for k = i + 1, …, j – 1 }
time/subproblem = O(j – i) = O(n)

Parenthesization

Task: Optimal evaluation of associative expression $A[0] \cdot A[1] \cdots A[n-1]$ (e.g., multiplying matrices)

Topological order:

increasing substring size

total time = $O(n^3)$

Final problem:

DP[0, n]

parent pointers to recover parenthesization

Parenthesization

```
// Matrix A[i] has dimension dims[i-1] x dims[i] for i = 1..n
MatrixChainOrder(int dims[])
{
    // length[dims] = n + 1
    n = dims.length - 1;
    // m[i,j] = Minimum number of scalar multiplications (i.e., cost)
    // needed to compute the matrix A[i]A[i+1]...A[j] = A[i..j]
    // The cost is zero when multiplying one matrix
    for (i = 1; i <= n; i++)
        m[i, i] = 0;

    for (len = 2; len <= n; len++) { // Subsequence lengths
        for (i = 1; i <= n - len + 1; i++) {
            j = i + len - 1;
            m[i, j] = MAXINT;
            for (k = i; k <= j - 1; k++) {
                cost = m[i, k] + m[k+1, j] + dims[i-1]*dims[k]*dims[j];
                if (cost < m[i, j]) {
                    m[i, j] = cost;
                    s[i, j] = k; // Index of the subsequence split that achieved minimal cost
                }
            }
        }
    }
}
```

Edit distance

Task: find the cheapest way to convert string x into y

Operations with cost:

- insert c
- delete c
- replace c with c'

Examples:

- Spell checking (typos)
- DNA comparison

Edit distance

Task: find the cheapest way to convert string x into y

Operations with cost:

- insert c
- delete c
- replace c with c'

If insert and delete cost 1 and replace costs ∞ is equivalent to **longest common subsequence**

HIEROGLYPHOLOGY
MICHAELANGELO

Edit distance

Task: find the cheapest way to convert string x into y

Operations with cost:

- insert c
- delete c
- replace c with c'

If insert and delete cost 1 and replace costs ∞ is equivalent to **longest common subsequence**

HIEROGLYPHOLOGY
MICHAELANGELO

Answer: HELLO

Edit distance

Task: find the cheapest way to convert string x into y

Subproblems:

c[i, j] = edit-distance(x[i :], y[j :]), $0 \le i \le |x|, 0 \le j \le |y|$

#subproblems = O(|x| |y|)

Guesses:

x[i] deleted, y[j] inserted, x[i] replaced with y[j]

#choices = 3

Recurrence: time/subproblem = O(1)

c(i , j)=min{

cost(delete x[i]) + c(i + 1, j), if i < |x|;

cost(insert y[j]) + c(i, j + 1), if j < |y|;

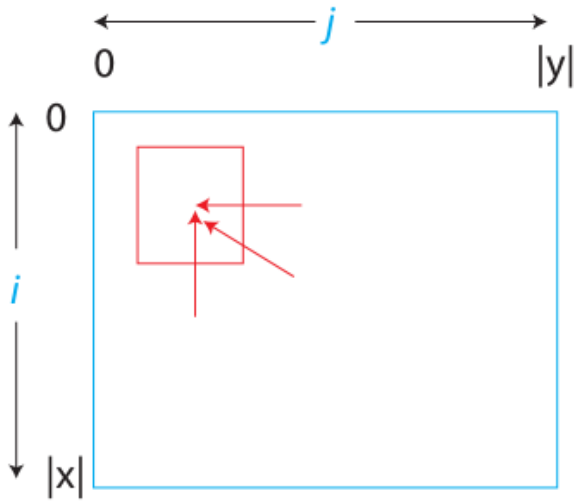
cost(replace x[i] with y[j]) + c(i + 1, j + 1), i < |x|, j < |y| }

Edit distance

Task: find the cheapest way to convert string x into y

Topological order:
DAG in 2D table

total time = O(|x| |y|)



Final problem:

c(0, 0)

parent pointers to recover the path

Edit distance

Example of traversal from right to left

```cpp
// A Naive recursive C++ program to find minimum number
// operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z)
{
   return min(min(x, y), z);
}

int editDist(string str1 , string str2 , int m ,int n)
{
    // If first string is empty, the only option is to
    // insert all characters of second string into first
    if (m == 0) return n;

    // If second string is empty, the only option is to
    // remove all characters of first string
    if (n == 0) return m;

    // If last characters of two strings are same, nothing
    // much to do. Ignore last characters and get count for
    // remaining strings.
    if (str1[m-1] == str2[n-1])
        return editDist(str1, str2, m-1, n-1);

    // If last characters are not same, consider all three
    // operations on last character of first string, recursively
    // compute minimum cost for all three operations and take
    // minimum of three values.
    return 1 + min ( editDist(str1,  str2, m, n-1),    // Insert
                     editDist(str1,  str2, m-1, n),   // Remove
                     editDist(str1,  str2, m-1, n-1) // Replace
                   );
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDist( str1 , str2 , str1.length(), str2.length());

    return 0;
}
```

Knapsack

Task: pack most value into the knapsack

- item i has size s_i and value v_i
- capacity constraint S
- choose a subset of items to max the total value subject to total size ≤ S

Knapsack

Task: maximize value given S, s_i , v_i

Subproblems:
value for prefix i: items 0, …, i – 1
#subproblems = O(n)

Guesses:
include item i or not
#choices = 2

Recurrence:
DP[i + 1] = max{ DP[i], v_i + DP[i] if $s_i \leq S$ }

Knapsack

Task: maximize value given S, s_i , v_i

Subproblems:
value for prefix i: items 0, …, i – 1
#subproblems = O(n)

Guesses:
include item i or not
#choices = 2

Recurrence:
DP[i + 1] = max{ DP[i], v_i + DP[i] if $s_i \leq S$ }

Not enough information to know whether item i fits.
How much is left?

Knapsack, version 1

Task: maximize value given S, s_i - integer, v_i

Subproblems:
value for prefix i (items 0, …, i – 1) given knapsack of size s
#subproblems = O(n S)

Guesses:
include item i or not
#choices = 2

Recurrence:
DP[i + 1, s] = max{ DP[i, s], v_i + DP[i, s - s_i] if $s_i \leq S$ }
Base-case: DP[0, 0] = 0; DP[0, s] = ∞, s = 1, …, S

time/subproblem = O(1)

Knapsack, version 1

Task: maximize value given S, s_i - integer, v_i

Topological order:

for i = 0, …, n – 1:

for s in 0, …, S:

total time = O(n S)

Final problem:

find maximal DP[n – 1, s], s = 0, …, S

parent pointers to recover the set of items

Knapsack, version 1

Task: maximize value given S, s_i - integer, v_i

Is O(n S) good running time?

Polynomial time = polynomial in input size

- Here O(n) if number S fits in type int
- O(n log S) is polynomial
- S is exponential in log S (not polynomial)

Pseudopolynomial time = polynomial in the problem size AND the numbers in input

Knapsack, version 2

Task: maximize value given S, s_i, v_i - integer

Subproblems:
size for prefix i (items 0, …, i – 1) given items of value v
#subproblems = O(n V)

Guesses:
include item i or not
#choices = 2

Recurrence:
DP[i + 1, v] = min{ DP[i, v], s_i + DP[i, v - v_i] if $v_i \leq v$ }
Base-case: DP[0, 0] = 0; DP[0, v] = ∞, v = 1, …, V

time/subproblem = O(1)

Knapsack, version 2

Task: maximize value given S, s_i, v_i - integer

Topological order:

for i = 0, …, n – 1:

for v in 0, …, V:

total time = O(n V)

Final problem:

find DP[n, v] ≤ S with maximal possible v

Knapsack, version 3

Task: maximize value given S, s_i, v_i

Subproblems:
value for prefix i (items 0, …, i – 1) given knapsack of size s (float)

Need to use floats as indices: hash tables (dictionaries)
DP[i][s]
#subproblems = 2^n in the worst case

Recurrence:
DP[i + 1][s] = max{ DP[i][s], v_i + DP[i][s - s_i] if $s_i \leq S$ }
Base-case: DP[0, 0] = 0; DP[0, s] = ∞

Bottom-up or memorized DP?

None is possible, subproblems are unknown in advance

General idea: forward recursion/reaching

Task: maximize value given S, s_i, v_i

Subproblems:
value for prefix i (items 0, …, i – 1) given knapsack of size s (float)

Create only reachable subproblems

Step 0:
DP[0][0] = 0

Step 1:
DP[1][0] = 0, DP[1][s_0] = v_0

Step 2:
From each position available at step 1 can add object 2

General idea: forward recursion/reaching

Task: maximize value given S, s_i, v_i

Subproblems:
value for prefix i (items 0, …, i – 1) given knapsack of size s (float)

Step i:

```
for s, v in DP[ i ].iteritems():
    for x in [ 0, 1 ]:
        if ( s + x s_i ) in DP[ i + 1 ]:
            v = DP[ i + 1 ][ s + x s_i ]
        else:
            v = -∞
        DP[ i + 1 ][ size + x s_i ] = max{  v, value + x v_i  }
```

General idea: forward recursion/reaching

Task: maximize value given S, s_i, v_i

Subproblems:
value for prefix i (items 0, …, i – 1) given knapsack of size s (float)

Problem: #subproblems is 2^n in the worst case

We need to throw away bad subproblems early!

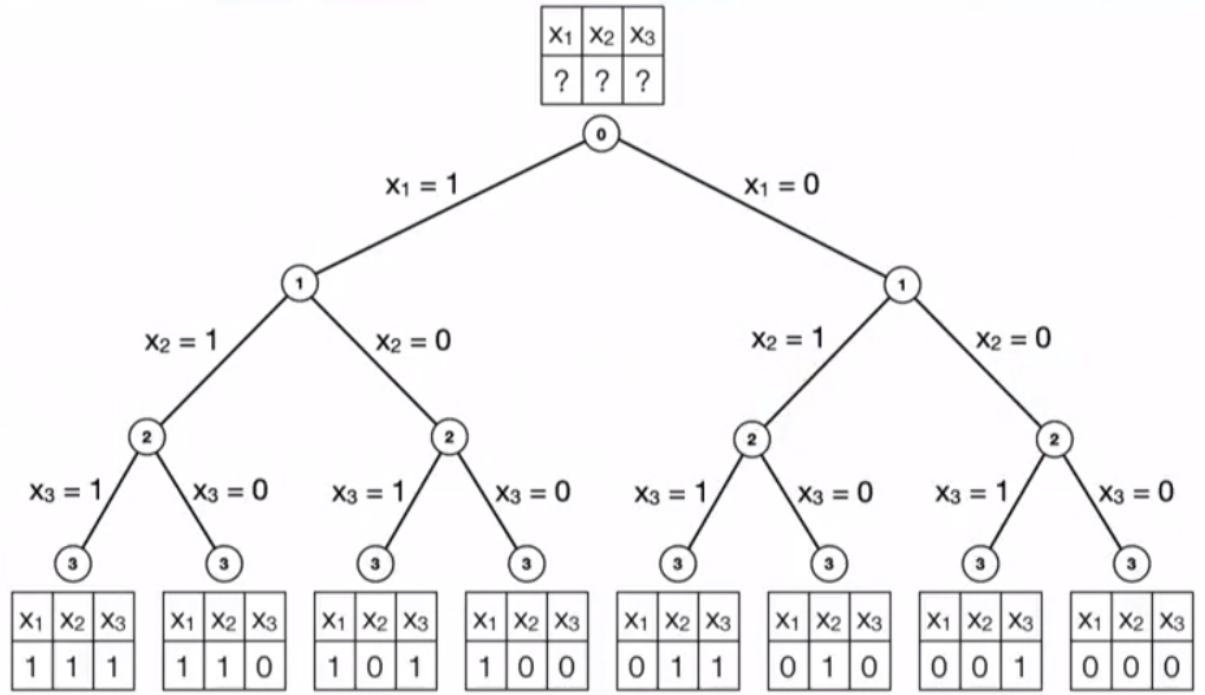
Branch-and-bound can do it!

General idea: branch-and-bound

Task: maximize value given S, s_i, v_i

Exhaustive search:

Branch-and-bound cuts the branches



General idea: branch-and-bound

Task: maximize value given S, s_i, v_i

Two iterative steps: branching and bounding

Branching: split the problem into a number of subproblems (like DP)

Bounding: find an optimistic estimate of the best solution of the subproblem

- upper bound for maximization
- lower bound for minimization

How to find a bound?

relaxation

Knapsack as integer program

Task: maximize value given S, s_i, v_i

Example: S = 10, 3 objects: $s_0 = 5$, $s_1 = 8$, $s_2 = 3$, $v_0 = 45$, $v_1 = 48$, $v_2 = 35$

Integer linear program (ILP):

max $45x_0 + 48x_1 + 35x_2$

s.t. $5x_0 + 8x_1 + 3x_2 \leq 10$

$x_i \in \{0, 1\}$

Relaxation for knapsack, version 1

Task: maximize value given S, s_i, v_i

Example: S = 10, 3 objects: $s_0 = 5$, $s_1 = 8$, $s_2 = 3$, $v_0 = 45$, $v_1 = 48$, $v_2 = 35$

What can we relax?

max $45x_0 + 48x_1 + 35x_2$

s.t. ~~$5x_0 + 8x_1 + 3x_2 \leq 10$~~

$x_i \in \{0, 1\}$

Relax capacity constraint!

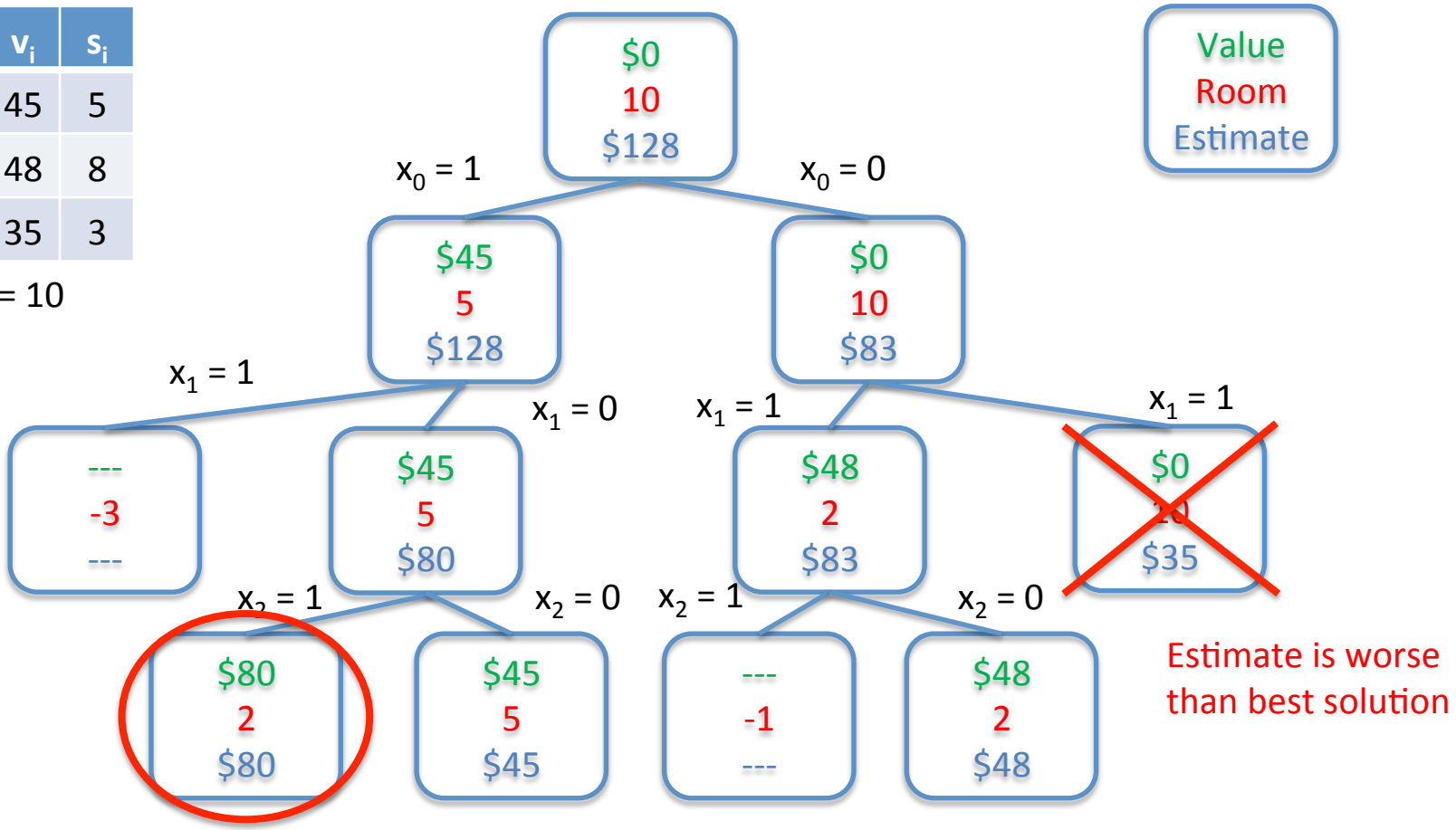
Pack everything ignoring capacity

Depth-first Branch-and-bound

Task: maximize value given S, s_i, v_i

i	v_i	s_i
0	45	5
1	48	8
2	35	3

S = 10



Relaxation for knapsack, version 2

Task: maximize value given S, s_i, v_i

Example: S = 10, 3 objects: $s_0 = 5$, $s_1 = 8$, $s_2 = 3$, $v_0 = 45$, $v_1 = 48$, $v_2 = 35$

What else can we relax?

max $45x_0 + 48x_1 + 35x_2$

s.t. $5x_0 + 8x_1 + 3x_2 \leq 10$

~~$x_i \in \{0, 1\}$~~ → $x_i \in [0, 1]$

Relax integrality constraint!

Imagine that items are possible to cut in pieces

Relaxation for knapsack, version 2

Task: maximize value given S, s_i, v_i

Example: S = 10, 3 objects: $s_0 = 5$, $s_1 = 8$, $s_2 = 3$, $v_0 = 45$, $v_1 = 48$, $v_2 = 35$

How do we solve the relaxation?

max $45x_0 + 48x_1 + 35x_2$

s.t. $5x_0 + 8x_1 + 3x_2 \leq 10$

$x_i \in [0, 1]$

$v_0 / s_0 = 9$; $v_1 / s_1 = 6$; $v_2 / s_2 = 11.7$

select items 2 and 0

Select ¼ of item 1

Estimation: 92 (version 1: 128; optimal value: 80)

Denote $y_i = x_i v_i$

$$\max \sum_i y_i$$

$$\text{s.t.} \sum_i y_i \frac{s_i}{v_i} \leq S$$

$$y_i \leq v_i$$

Sort s_i / v_i increasing

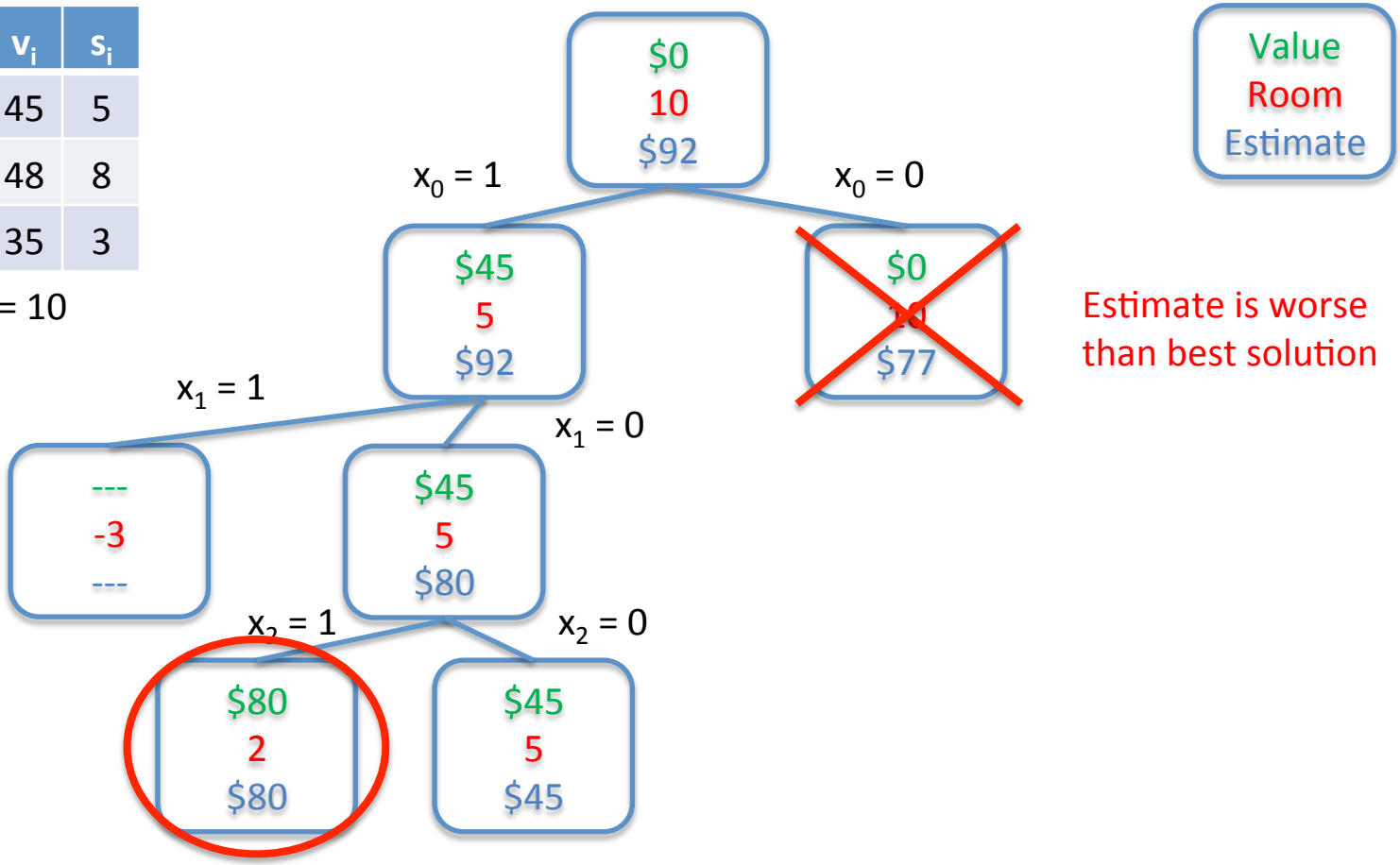
Take as much as possible

Depth-first Branch-and-bound

Task: maximize value given S, s_i, v_i

i	v_i	s_i
0	45	5
1	48	8
2	35	3

S = 10



Value
Room
Estimate

Estimate is worse than best solution

Branch-and-bound: search strategies

- Depth-first search

  go deep until full configuration

- Best-first search

  select the node with the best estimation

- Limited discrepancy search

  trust a greedy heuristic

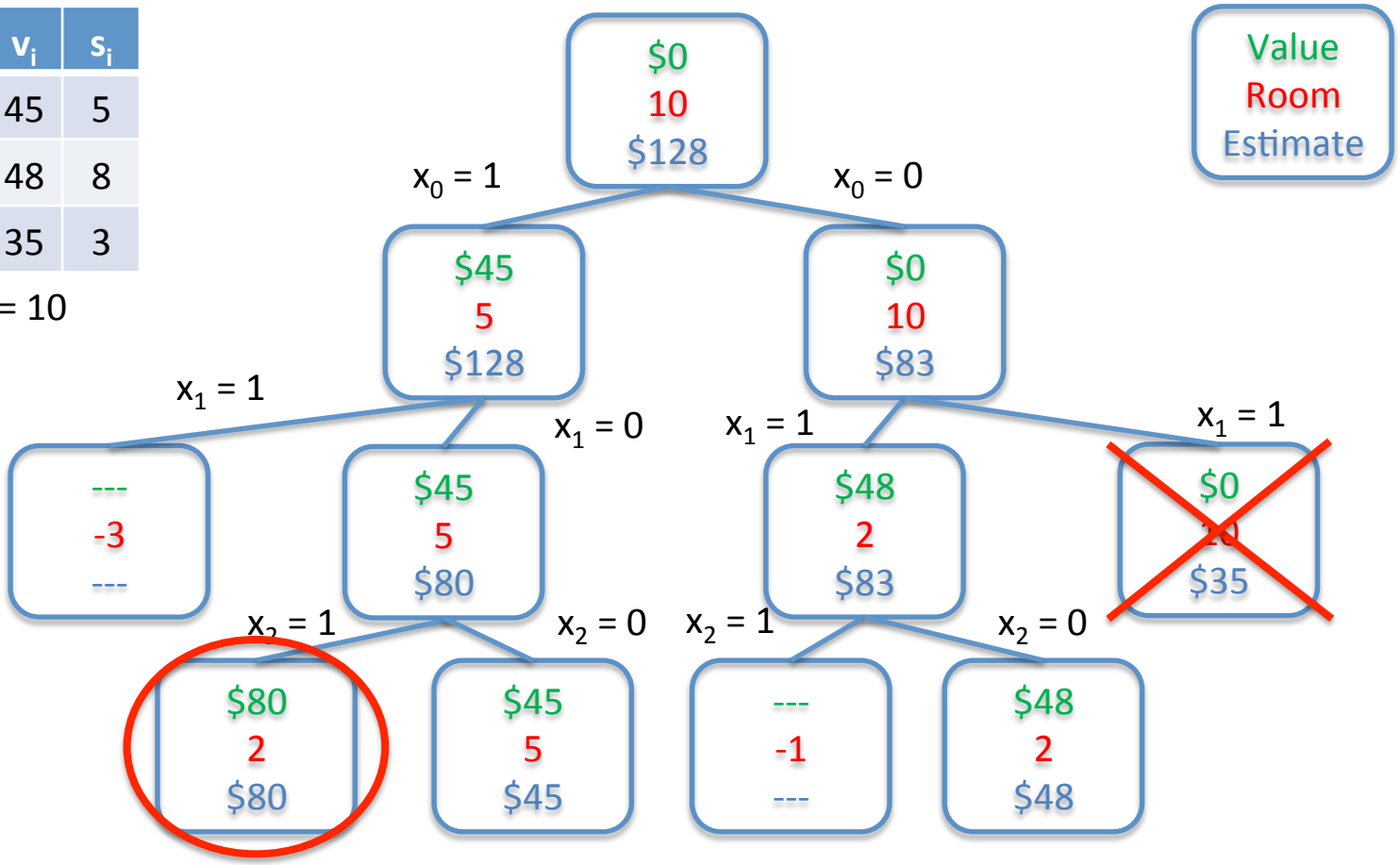
Properties: when prunes? memory efficiency?

Depth-first Branch-and-bound

Task: maximize value given S, s_i, v_i

i	v_i	s_i
0	45	5
1	48	8
2	35	3

S = 10



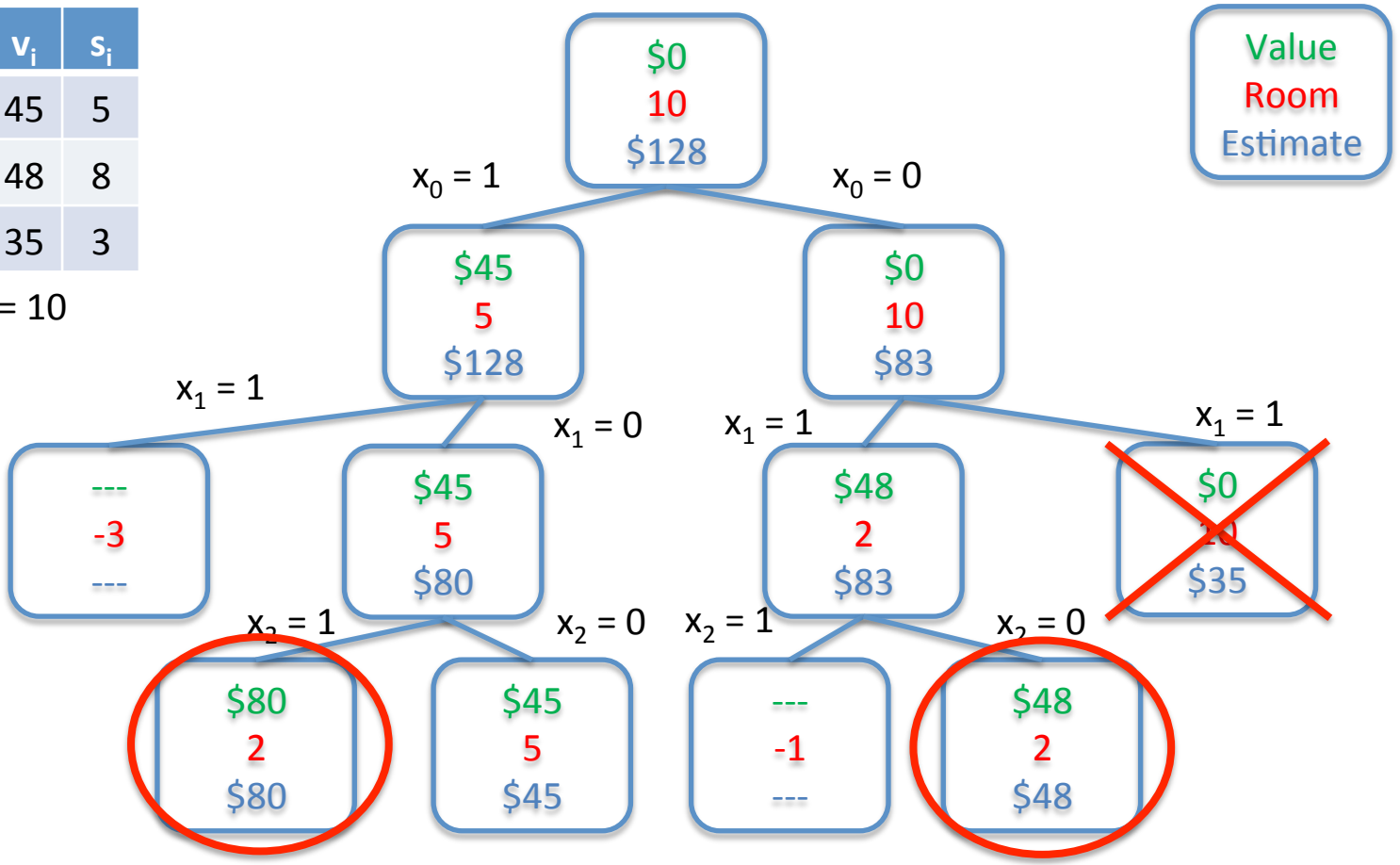
Best-first Branch-and-bound

Task: maximize value given S, s_i, v_i

Example: S = 10, 3 objects: $s_0 = 5$, $s_1 = 8$, $s_2 = 3$, $v_0 = 45$, $v_1 = 48$, $v_2 = 35$

i	v_i	s_i
0	45	5
1	48	8
2	35	3

S = 10



Limited discrepancy search

Task: maximize value given S, s_i, v_i

Assume that a good heuristic is available

- it makes very few mistakes
- the search tree is binary
- following the heuristics means branching left
- branching right means the heuristic was wrong

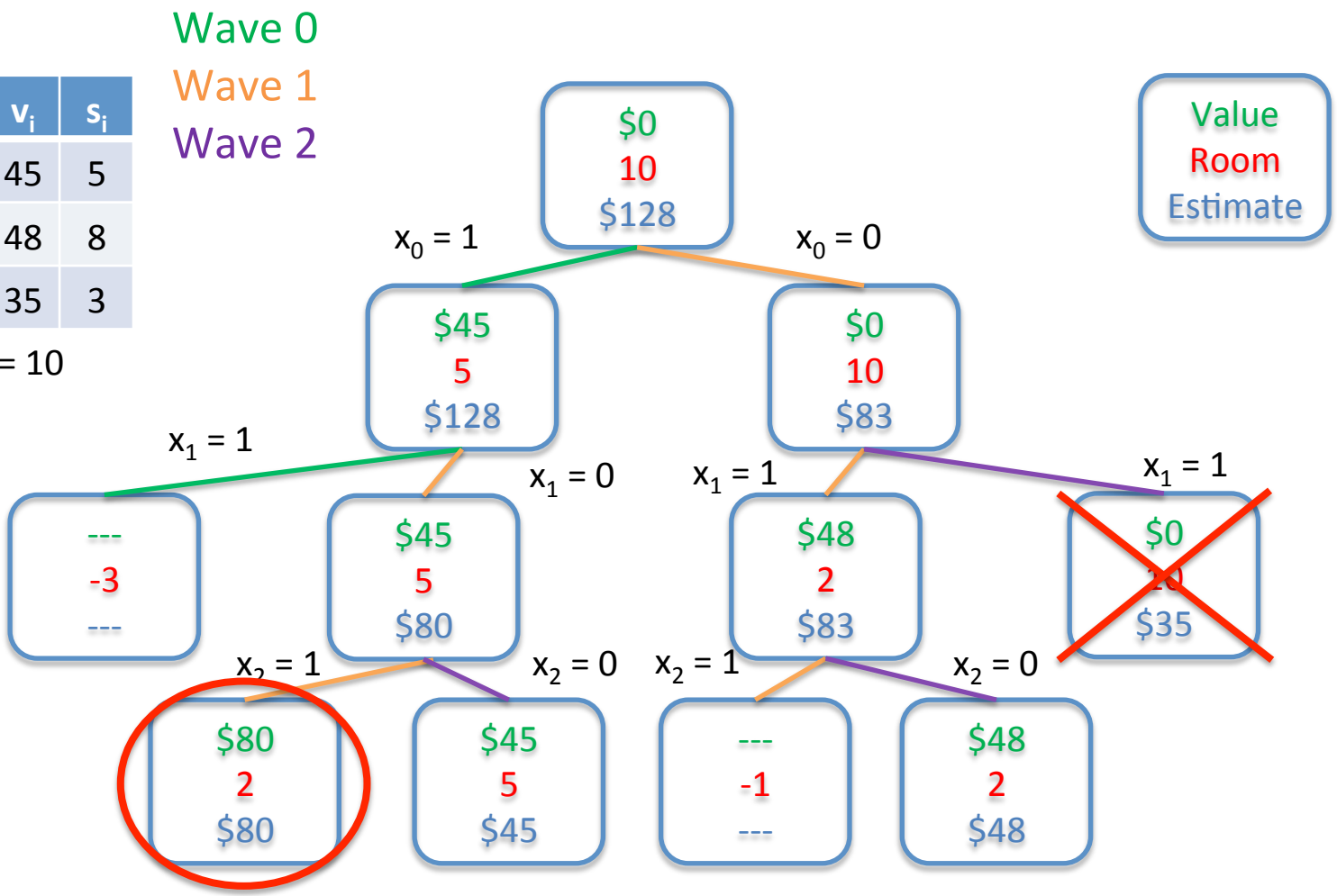
Explore configurations with less mistakes first

Limited discrepancy search

Task: maximize value given S, s_i, v_i

i	v_i	s_i
0	45	5
1	48	8
2	35	3

S = 10



Branch-and-bound: search strategies

- Depth-first search

  prunes when finds a new node worse than found solution

  memory O(n)

- Best-first search

  prunes when all the nodes are worse than found solution

  memory O(2^n) in the worst case

- Limited discrepancy search

  prunes when finds a new node worse than found solution

  trade-off between memory and computation

Properties: when prune? memory efficiency?

More dynamic programming

- Guitar fingering
- Hardwood floor (parquet)
- Tetris
- Blackjack
- Super Mario Bros