CH08-320201

Algorithms and Data Structures

ADS

Lecture 25

Dr. Kinga Lipskoch

Spring 2019

Linear Programming[1]

- Important tool for optimal allocation of scarce resources, among a number of competing activities.
- Powerful and general problem-solving method.
- Applications:
  - Computer science: Compiler register allocation, data mining.
  - Electrical engineering: VLSI design, optimal clocking.
  - Economics: Equilibrium theory, two-person zero-sum games.
  - Environment: Water quality management.
  - Logistics: Supply-chain management, Berlin airlift.
  - Manufacturing: Production line balancing, cutting stock.
  - Telecommunication: Network design, Internet routing.

[1] Source of slides: Kevin Wayne: Algorithms and Data Structures, Spring 2004, Princeton University

Brewery Problem: A Toy LP Example

Small brewery produces ale and beer.

- Production limited by scarce resources: corn, hops, malt.
- Recipes for ale and beer require different proportions of resources.

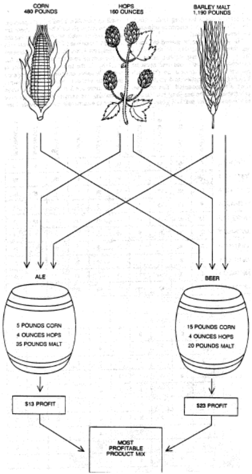
Beverage	Corn (pounds)	Hops (ounces)	Malt (pounds)	Profit ($)
Ale	5	4	35	13
Beer	15	4	20	23
Quantity	480	160	1190	

How can brewer maximize profits?

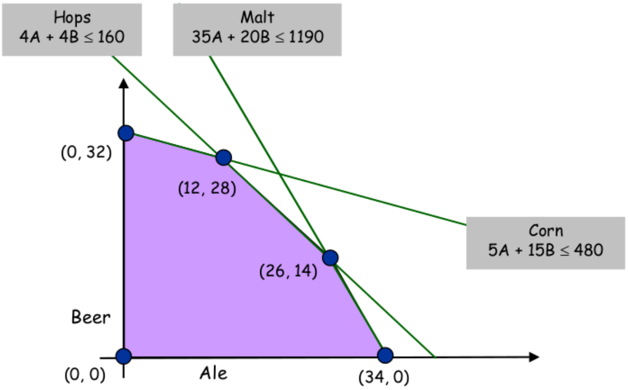
- Devote all resources to ale: 34 barrels of ale $\to$ \$442.
- Devote all resources to beer: 32 barrels of beer $\to$ \$736.
- 7.5 barrels of ale, 29.5 barrels of beer $\to$ \$776.
- 12 barrels of ale, 28 barrels of beer $\to$ \$800.

Brewery Problem

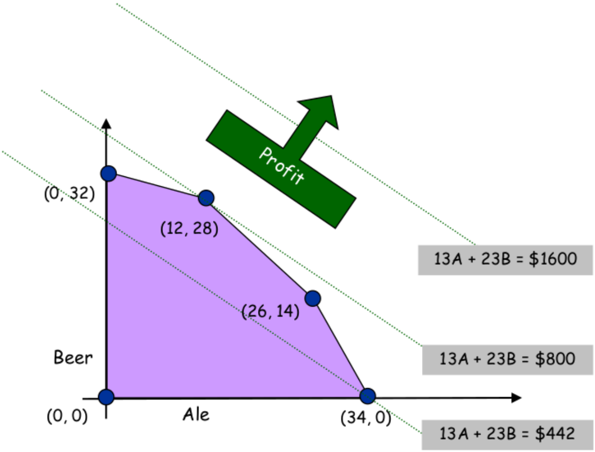
	Ale		Beer			
max	$13A$	+	$23B$			Profit
s. t.	$5A$	+	$15B$	$\leq$	480	Corn
	$4A$	+	$4B$	$\leq$	160	Hops
	$35A$	+	$20B$	$\leq$	1190	Malt
	A	,	B	$\geq$	0	



Brewery Problem: Feasible Region

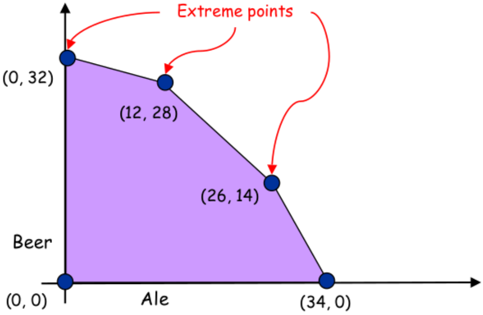


Brewery Problem: Objective Function



Brewery Problem: Geometry

Observation: Regardless of objective function coefficients, an optimal solution occurs at an extreme point.



Linear Programming: Standard Form

Standard form:

- Input: real numbers c_j, b_i, a_{ij}.
- Output: real numbers x_j.
- $n = \#$ nonnegative variables, $m = \#$ constraints.
- Maximize linear objective function subject to linear inequalities.

$$\begin{array}{llrcll} \text{(P)} & \max & \sum_{j=1}^{n} c_j x_j & & & \\ & \text{s.t.} & \sum_{j=1}^{n} a_{ij} x_j & = & b_i & 1 \le i \le m \\ & & x_j & \ge & 0 & 1 \le j \le n \end{array}$$

$$\begin{array}{llrcl} \text{(P)} & \max & c^T x & & \\ & \text{s.t.} & Ax & = & b \\ & & x & \ge & 0 \end{array}$$

Linear: No x^2 , xy, $\arccos(x)$, etc.
Programming: Planning (term predates computer programming).

Brewery Problem: Converting to Standard Form

Original input:

$$\begin{array}{lrcrcr} \max & 13A & + & 23B & & \\ \text{s.t.} & 5A & + & 15B & \leq & 480 \\ & 4A & + & 4B & \leq & 160 \\ & 35A & + & 20B & \leq & 1190 \\ & A & , & B & \geq & 0 \end{array}$$

Standard form:

- Add slack variable for each inequality.
- Now a 5-dimensional problem.

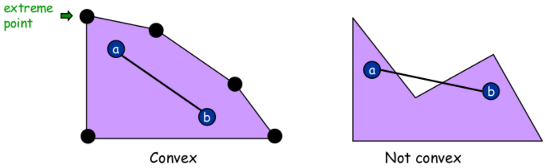
$$\begin{array}{lrcrcccccccr} \max & 13A & + & 23B & & & & & & & & \\ \text{s.t.} & 5A & + & 15B & + & S_C & & & & & = & 480 \\ & 4A & + & 4B & & & + & S_H & & & = & 160 \\ & 35A & + & 20B & & & & & + & S_M & = & 1190 \\ & A & , & B & , & S_C & , & S_H & , & S_M & \geq & 0 \end{array}$$

Geometry (1)

Geometry:

- Inequalities : halfplanes (2D), hyperplanes.
- Bounded feasible region: convex polygon (2D), (convex) polytope.

Convex: if a and b are feasible solutions, then so is $(a+b)/2$.
Extreme point: feasible solution x that cannot be written as $(a+b)/2$ for any two distinct feasible solutions a and b.



Geometry (2)

Extreme point property: If there exists an optimal solution to (P), then there exists one that is an extreme point.

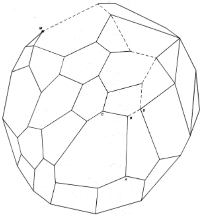
- Only need to consider finitely many possible solutions.

Challenge: Number of extreme points can be exponential.

- Consider n-dimensional hypercube.

Greedy: Local optima are global optima.

- Extreme point is optimal if no neighboring extreme point is better.



Simplex Algorithm

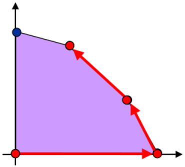
Simplex algorithm: (George Dantzig, 1947)

- Developed after WWII in response to logistical problems.
- Used for 1948 Berlin airlift.

Generic algorithm:

- Start at some extreme point.
- Pivot from one extreme point to a neighboring one. (never decrease objective function)
- Repeat until optimal.

How to implement? Linear algebra.



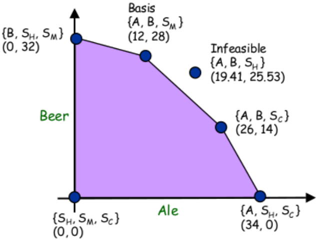
Simplex Algorithm: Basis

Basis: Subset of m of the n variables.

Basic feasible solution (BFS): Set $n - m$ nonbasic variables to 0, solve for remaining m variables.

- Solve m equations in m unknowns.
- If unique and feasible solution $\rightarrow$ BFS.
- BFS corresponds to extreme point.
- Simplex only considers BFS.

$$
\begin{array}{llllllll}
\max & 13A & + & 23B & & & & \\
\text{s.t.} & 5A & + & 15B & + S_C & & & = 480 \\
 & 4A & + & 4B & & + S_H & & = 160 \\
 & 35A & + & 20B & & & + S_M & = 1190 \\
 & A & , & B & , S_C & , S_H & , S_M & \geq 0
\end{array}
$$



Simplex Algorithm: Pivot 1 (1)

max Z subject to

$$\begin{array}{rcrcrcrcrcrcr}
13A & + & 23B & & & & & & & - & Z & = & 0 \\
\hline
5A & + & 15B & + & S_C & & & & & & & = & 480 \\
4A & + & 4B & & & + & S_H & & & & & = & 160 \\
35A & + & 20B & & & & & + & S_M & & & = & 1190 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{S_C, S_H, S_M\}$
$A = B = 0$
$Z = 0$
$S_C = 480$
$S_H = 160$
$S_M = 1190$

Substitute: $B = 1/15\,(480 - 5A - S_C)$

max Z subject to

$$\begin{array}{rcrcrcrcrcrcr}
\frac{16}{3}A & & & - & \frac{23}{15}S_C & & & & & - & Z & = & -736 \\
\hline
\frac{1}{3}A & + & B & + & \frac{1}{15}S_C & & & & & & & = & 32 \\
\frac{8}{3}A & & & - & \frac{4}{15}S_C & + & S_H & & & & & = & 32 \\
\frac{85}{3}A & & & - & \frac{4}{3}S_C & & & + & S_M & & & = & 550 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}$$

Basis = $\{B, S_H, S_M\}$
$A = S_C = 0$
$Z = 736$
$B = 32$
$S_H = 32$
$S_M = 550$

Simplex Algorithm: Pivot 1 (2)

max Z subject to

$$
\begin{array}{rcrcccccccl}
13A & + & 23B & & & & & - & Z & = & 0 \\
\hline
5A & + & 15B & + & S_C & & & & & = & 480 \\
4A & + & 4B & & & + & S_H & & & = & 160 \\
35A & + & 20B & & & & & + & S_M & = & 1190 \\
A & , & B & , & S_C & , & S_H & , & S_M & \geq & 0
\end{array}
$$

Basis = $\{S_C, S_H, S_M\}$
$A = B = 0$
$Z = 0$
$S_C = 480$
$S_H = 160$
$S_M = 1190$

Why pivot on column 2?

- Each unit increase in B increases objective value by \$23.
- Pivoting on column 1 also OK.

Why pivot on row 2?

- Preserves feasibility by ensuring $RHS \geq 0$.
- Minimum ratio rule: $\min\{480/15, 160/4, 1190/20\}$.

Simplex Algorithm: Pivot 2

max Z subject to

$$\begin{array}{rcl} \frac{16}{3}A - \frac{23}{15}S_C - Z &=& -736 \\ \hline \frac{1}{3}A + B + \frac{1}{15}S_C &=& 32 \\ \frac{8}{3}A - \frac{4}{15}S_C + S_H &=& 32 \\ \frac{85}{3}A - \frac{4}{3}S_C + S_M &=& 550 \\ A, B, S_C, S_H, S_M &\geq& 0 \end{array}$$

Basis = $\{B, S_H, S_M\}$
$A = S_C = 0$
$Z = 736$
$B = 32$
$S_H = 32$
$S_M = 550$

Substitute: $A = 3/8\,(32 + 4/15\,S_C - S_H)$

max Z subject to

$$\begin{array}{rcl} -S_C - 2S_H - Z &=& -800 \\ \hline B + \frac{1}{10}S_C + \frac{1}{8}S_H &=& 28 \\ A - \frac{1}{10}S_C + \frac{3}{8}S_H &=& 12 \\ -\frac{25}{6}S_C - \frac{85}{8}S_H + S_M &=& 110 \\ A, B, S_C, S_H, S_M &\geq& 0 \end{array}$$

Basis = $\{A, B, S_M\}$
$S_C = S_H = 0$
$Z = 800$
$B = 28$
$A = 12$
$S_M = 110$

Simplex Algorithm: Optimality

When to stop pivoting?

- If all coefficients in top row are non-positive.

Why is resulting solution optimal?

- Any feasible solution satisfies system of equations in tableaux.
  - in particular: $Z = 800 - S_C - 2S_H$
- Thus, optimal objective value $Z^* \leq 800$ since $S_C, S_H \geq 0$.
- Current BFS has value 800 $\rightarrow$ optimal.

max Z subject to

$$
\begin{array}{rcrcrcrcrcrcr}
 & & & - & S_C & - & 2\,S_H & & & - & Z & = & -800 \\
\hline
 & & B & + & \frac{1}{10}\,S_C & + & \frac{1}{8}\,S_H & & & & & = & 28 \\
A & & & - & \frac{1}{10}\,S_C & + & \frac{3}{8}\,S_H & & & & & = & 12 \\
 & & & - & \frac{25}{6}\,S_C & - & \frac{85}{8}\,S_H & + & S_M & & & = & 110 \\
A & , & B & , & S_C & , & S_H & , & S_M & & & \geq & 0
\end{array}
$$

Basis = $\{A, B, S_M\}$
$S_C = S_H = 0$
$Z = 800$
$B = 28$
$A = 12$
$S_M = 110$

Simplex Algorithm: Issues

Remarkable property: In practice, simplex algorithm typically terminates in at most $2(m+n)$ pivots.

- No polynomial pivot rule known.
- Most pivot rules known to be exponential in worst-case.

Issues: Which neighboring extreme point?

Degeneracy: New basis, same extreme point.

- "Stalling" is common in practice.

Cycling: Get stuck by cycling through different bases that all correspond to same extreme point.

- Does not occur in the wild.
- Bland's least index rule $\rightarrow$ finite # of pivots.

Backtracking: Motivation[2]

Example Sudoku solving

							1	2
				3	5			
			6				7	
7						3		
			4			8		
1								
			1	2				
	8						4	
	5					6		

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

[2] Source of slides: Steven Skiena, Lecture slides, Stony Brook University

Solving Sudoku

- Solving Sudoku puzzles involves a form of exhaustive search of possible configurations.
- However, exploiting constraints to rule out certain possibilities for certain positions enables us to prune the search to the point people can solve Sudoku by hand.
- Backtracking is a general algorithm which can be used to implement exhaustive search programs correctly and efficiently.

Backtracking Technique

- Backtracking is a systematic method to iterate through all the possible configurations of a search space.
- It is a general algorithm/technique which must be customized for each individual application.
- In the general case, we will model our solution as a vector $a = (a_1, a_2, ..., a_n)$, where each element a_i is selected from a finite ordered set S_i.
- Such a vector might represent an arrangement where a_i contains the i^{th} element of the permutation.
- Or the vector might represent a given subset S, where a_i is true if and only if the i^{th} element of the universe is in S.

The Idea of Backtracking

- At each step in the backtracking algorithm, we start from a given partial solution, $a = (a_1, a_2, ..., a_k)$, and try to extend it by adding another element at the end.
- After extending it, we must test whether what we have so far is a solution.
- If not, we must then check whether the partial solution is still potentially extendible to some complete solution.
- If so, recur and continue. If not, we delete the last element from a and try another possibility for that position, if one exists.

Recursive Backtracking

```
Backtrack(a, k)
  if a is a solution
    print(a)
  else {
    k = k +1
    compute S[k]
    while S[k] != empty do
      a[k] = an element in S[k]
      S[k] = S[k] - a[k]
      Backtrack(a, k)
  }
```

Backtracking and DFS

- Backtracking is just depth-first search on an implicit graph of configurations.
- Backtracking can easily be used to iterate through all subsets or permutations of a set.
- Backtracking ensures correctness by enumerating all possibilities.
- For backtracking to be efficient, we must prune the search space.

Implementation

```
bool finished = FALSE; /* all solutions? */
backtrack(int a[], int k, data input) {
  int c[MAXCANDIDATES]; /* cand. next pos. */
  int ncandidates; /* next pos. cand. count */
  int i; /* counter */
  if (is_a_solution(a, k, input))
    process_solution(a, k, input);
  else {
    k = k+1;
    construct_candidates(a, k, input, c,
      &ncandidates);
    for (i=0; i<ncandidates; i++) {
      a[k] = c[i];
      backtrack(a, k, input);
      if (finished) return; /* term. early */
    }}}
```

Is a Solution?

- `is_a_solution(a, k, input)`
- This boolean function tests whether the first `k` elements of vector `a` are a complete solution for the given problem.
- The last argument, `input`, allows us to pass general information into the routine.

Construct Candidates

- `construct_candidates(a, k, input, c, &ncandidates);`
- This routine fills an array `c` with the complete set of possible candidates for the `k`th position of `a`, given the contents of the first `k` -1 positions.
- The number of candidates returned in this array is denoted by `ncandidates`.

Process Solution

- `process_solution(a, k)`
- This routine prints, counts, or somehow processes a complete solution once it is constructed.
- Backtracking ensures correctness by enumerating all possibilities. It ensures efficiency by never visiting a state more than once.
- Because a new candidates array `c` is allocated with each recursive procedure call, the subsets of not-yet-considered extension candidates at each position will not interfere with each other.

Constructing all Subsets (1)

- How many subsets are there of an n-element set?
- To construct all 2^n subsets, set up an array/vector of n elements, where the value of a_i is either true or false, signifying whether the i^{th} item is or is not in the subset.
- To use the notation of the general backtrack algorithm, $S_k = (true, false)$, and v is a solution whenever $k \geq n$.
- What order will this generate the subsets of $\{1, 2, 3\}$?
  $(1) \rightarrow (1, 2) \rightarrow (1, 2, 3) \rightarrow$
  $(1, 2, -) \rightarrow (1, -) \rightarrow (1, -, 3) \rightarrow$
  $(1, -, -) \rightarrow (1, -) \rightarrow (1) \rightarrow$
  $(-) \rightarrow (-, 2) \rightarrow (-, 2, 3) \rightarrow$
  $(-, 2, --) \rightarrow (-, -) \rightarrow (-, -, 3) \rightarrow$
  $(-, -, -) \rightarrow (-, -) \rightarrow (-) \rightarrow ()$

Constructing all Subsets (2)

- We can construct the 2^n subsets of n items by iterating through all possible 2^n length$-n$ vectors of *true* or *false*, letting the i^{th} element denote whether item i is or is not in the subset.
- Using the notation of the general backtrack algorithm, $S_k = (\mathit{true}, \mathit{false})$, and a is a solution whenever $k \geq n$.

Constructing all Subsets (3)

```
is_a_solution(int a[], int k, int n) {
  return (k == n); /* is k == n? */
}
construct_candidates(int a[], int k, int n, int
    c[], int *ncandidates) {
  c[0] = TRUE;
  c[1] = FALSE;
  *ncandidates = 2;
}
process_solution(int a[], int k) {
  int i; /* counter */
  print("(");
  for (i=1; i<=k; i++)
    if (a[i] == TRUE)
      print(")");
}
```

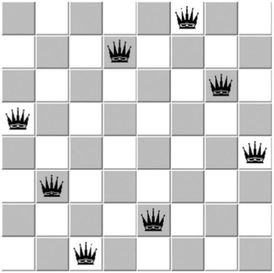
Main Routine: Subsets

- Finally, we must instantiate the call to backtrack with the corresponding arguments.

```
generate_subsets(int n) {
  int a[NMAX]; /* solution vector */
  backtrack(a, 0, n);
}
```

The Eight-Queens Problem

- The eight queens problem is a classical puzzle of positioning eight queens on an 8×8 chessboard such that no two queens threaten each other.
- This a classical textbook backtracking problem.



Eight Queens: Representation

- What is concise, efficient representation for an n-queens solution, and how big must it be?
- Since no two queens can occupy the same column, we know that the n columns of a complete solution must form a permutation of n.
- By avoiding repetitive elements, we reduce our search space to just $8! = 40,320$ quick for any reasonably fast machine.
- The critical routine is the candidate constructor.
- We repeatedly check whether the k^{th} square on the given row is threatened by any previously positioned queen.
- If so, we move on, but if not we include it as a possible candidate.
- Algorithm can find the $365,596$ solutions for $n = 14$ in minutes.