Computing for Economists

Introduction to Algorithms & Combinatorial Optimization

Toren Fronsdal and Ruby Zhang

Harvard University

What is an Algorithm?

- A computational procedure that takes an **input** and produces an **output** (value or set of values)
- A correct algorithm finishes computing in **finite time** and outputs the **correct solution** for all inputs
  - If you can control error rate $\rightarrow$ algorithm can be approximate and still helpful
- An algorithm is the procedure that achieves a desired input/output relationship of a computational problem
  - A problem **instance** specifies the input data

Resources

The module draws on the following resources:

- CS161: Design and Analysis of Algorithms
- “Algorithm Design” by Jon Kleinberg and Éva Tardos
- “Introduction to Algorithms” by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
- “Combinatorial Optimization: Algorithms and Complexity” by Christos H. Papadimitriou and Kenneth Steiglitz

Other helpful resources:

- “Network Flows” by Ravindra Ahuja, Thomas Magnanti, and James Orlin
- “Combinatorial Auctions” by Peter Cramton, Yoav Shoham, and Richard Steinberg

Table of Contents

1. Computational Complexity

2. Algorithmic Design

3. Combinatorial Optimization

Computational Complexity

Example: Matching Markets

How do we think about allocation in markets without prices but with preferences?

Stable Matching Problem
Given an ordered list of preferences for two equal sides (e.g. Firms and Workers), create *stable* pairwise matches

- A match is *stable* if no pair of Firm and Worker would prefer to switch matches

This economics problem can be solved with an *algorithm*

- Deferred Acceptance Algorithm (Gale and Shapley, 1962)[1]
- E.g., hospital residency matches, organ matches, school choice

[1] Subject of 2012 Nobel Prize in Economics!

Gale-Shapley Algorithm

1. Start with all Firms and Workers as unmatched
2. All Workers propose to their most preferred Firm which has not rejected them yet
3. Firms accept the most preferred Worker proposal and reject the rest (if it is preferred to the current match)
4. Start another round of proposals from Workers to Firms which have not rejected them
5. Iterate until all Workers and Firms are matched

Gale-Shapley is mathematically proven to arrive at some *stable* solution[2], but how *fast* is it?

[2] There are many possible stable matches, the solution is optimal for the side that proposes.

Gale-Shapley Running Time

Suppose we have n Workers and n Firms, and each proposal takes one unit of time

- What is the **best-case** scenario?
  - In Round 1, each Worker proposes to a different Firm and is accepted $\rightarrow$ time n
- What is the **worst-case** scenario?
  - The time cannot exceed every Worker proposing to every Firm $\rightarrow$ time $n \times n = n^2$
- What is the **average-case** scenario?
  - Average over all possible permutations of preferences and proposal orders for n Workers and Firms $\rightarrow$ time $n \log n$

Running Time Notation

- The goal of an algorithm is to be **efficient** and (almost) correct
- Given an input (e.g. points, vertices and edges) of size n, how **asymptotically** efficient is the algorithm?
  - $O(f(n))$: upper bound (i.e. function grows no faster than)
    - This is the most common notation
  - $\Theta(f(n))$: tight bound (i.e. function grows precisely at)
  - $\Omega(f(n))$: lower bound (i.e. function grows at least as fast as)
- "Little-o" notation of $o(f(n))$ and $\omega(f(n))$ are upper and lower bounds which are not asymptotically tight
- Distribution over inputs provides average-case running-time
  - For many problems, we can characterize best-case and worst-case running time

$\Rightarrow$ Gale-Shapley algorithm is $\Omega(n)$ and $O(n^2)$

Example: Search on a Sorted List

Another example is comparing different methods of searching for a target in a sorted list:

- Linear Search: Scanning element one by one
  - $O(n)$ time since time scales linearly with number of elements
- Binary Search: Compare target to middle element of array and if target is smaller (larger), search in the left (right) half
  - $O(\log n)$ time since the list is halved, $\log_2$

Complexity notation considers the highest order term and does not consider constants

- If the time is fixed, then it is $O(1)$

Time Comparisons

f(n)	n = 10	n = 100	n = 1000	n = 10,000	n = 100,000
n	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec
$n \log_2 n$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	2 sec
n^2	< 1 sec	< 1 sec	1 sec	2 min	3 hours
n^3	< 1 sec	1 sec	18 min	12 days	32 years
1.5^n	< 1 sec	12,892 years	$> 10^{25}$ years	$> 10^{25}$ years	$> 10^{25}$ years
2^n	< 1 sec	10^{17} years	$> 10^{25}$ years	$> 10^{25}$ years	$> 10^{25}$ years
$n!$	4 sec	10^{25} years	$> 10^{25}$ years	$> 10^{25}$ years	$> 10^{25}$ years

Table 1: Running times for a processor performing a million high-level instructions per second. Source.

P vs. NP

Three classes of problems:

- Class P problems can be **solved in polynomial time** $\rightarrow$ bounded by $O(n^k)$ time for some constant k
- Class NP problems can **verify** a solution **in polynomial time**
  - $P \subseteq NP$ but whether or not it's a proper subset is a famous open question (i.e. nondeterministic polynomial time)
- Class NP-complete (NPC) problems are NP and are as "hard" as any problem in NP
  - If any NPC problem can be solved in polynomial time $\rightarrow P = NP$
  - Believed (though not proved) to be intractable

General class of NP-Hard problems $\rightarrow$ as hard as the hardest problems in NP, but do not need to be in NP

NP-Hard Problems

What if your problem is *NP*-hard?

- Solve special instances of the problem
- Solve an approximation to the problem
- Solve the problem in exponential time (only for small n)

Example: Generalized Assignment Problem (GAP)

The following problem is **NP-Hard**!

Generalized Assignment Problem
Given tasks and agents where costs & profits are agent-task specific and agents have cost budgets, assign tasks to agents to maximize profit.

- Single agent reduces to knapsack problem
- All budgets and costs are one $\rightarrow$ assignment problem, solved with Hungarian algorithm (polynomial time)

GAP as Integer Linear Program

Framed as an integer linear program (NP-Hard):

$$\max_{x_{ij} \in \{0,1\}} \sum_{j=1}^{m} \sum_{i=1}^{n} p_{ij} x_{ij} \text{ s.t.}$$

$$\sum_{j=1}^{m} c_{ij} x_{ij} \leq b_i \qquad i = \{1, ..., n\}$$

$$\sum_{i=1}^{n} x_{ij} \leq 1 \qquad j = \{1, ..., m\}$$

where

- n agents represented by i, m tasks represented by j
- Agent i has budget b_i
- Task j completed by agent i costs c_{ij} and earns profit p_{ij}
- Each task is only assigned to one agent

Algorithmic Design

Example: Shortest Path Problem

Given a transportation network, what are the lowest cost/time paths between any two destinations?

- ...welfare of U.S. interstate highway system (Allen and Arkolakis, 2014)
- ...in the case of railroad networks (Donaldson, 2018)
- ...trade routes in ancient Assyria (Barjamovic et al., 2019)
- ...when calculating trade costs and flows in shipping (Brancaccio et al., 2020)

and even the “distance” in production networks with input-output data (Carvalho and Voigtländer, 2015)

Shortest Path Problem

Formal Problem Statement

Given a graph $G(V, E)$ with vertices V and positive-weighted edges E, what is the shortest path between two nodes o (origin) and d (destination)?

- Shortest path exhibits optimal substructure[3]
- Common problem in networks and urban/spatial economics (edge weights can be distance, cost, time, etc.)

How do we solve this problem with an *algorithm*?

- Dijkstra's, Floyd-Warshall, Bellman-Ford

[3] Shortest path is a polynomial time problem whereas longest path is NP-hard! The latter does not exhibit optimal substructure.

Algorithm Paradigms

Paradigms of Algorithm Design:

- Greedy algorithms
- Divide and conquer
- Dynamic programming

Greedy Algorithms

An algorithm is *greedy* if at each step, it makes a locally optimal decision without keeping track of the global solution

- Goal: converge to global solution in a reasonable time
- Ideal for problems with optimal substructure (i.e. optimal global solution is also optimal solution for all sub-problems)
- For many problems, will not find the global optimum since it's not an exhaustive search
- If close to correct, much faster than dynamic programming

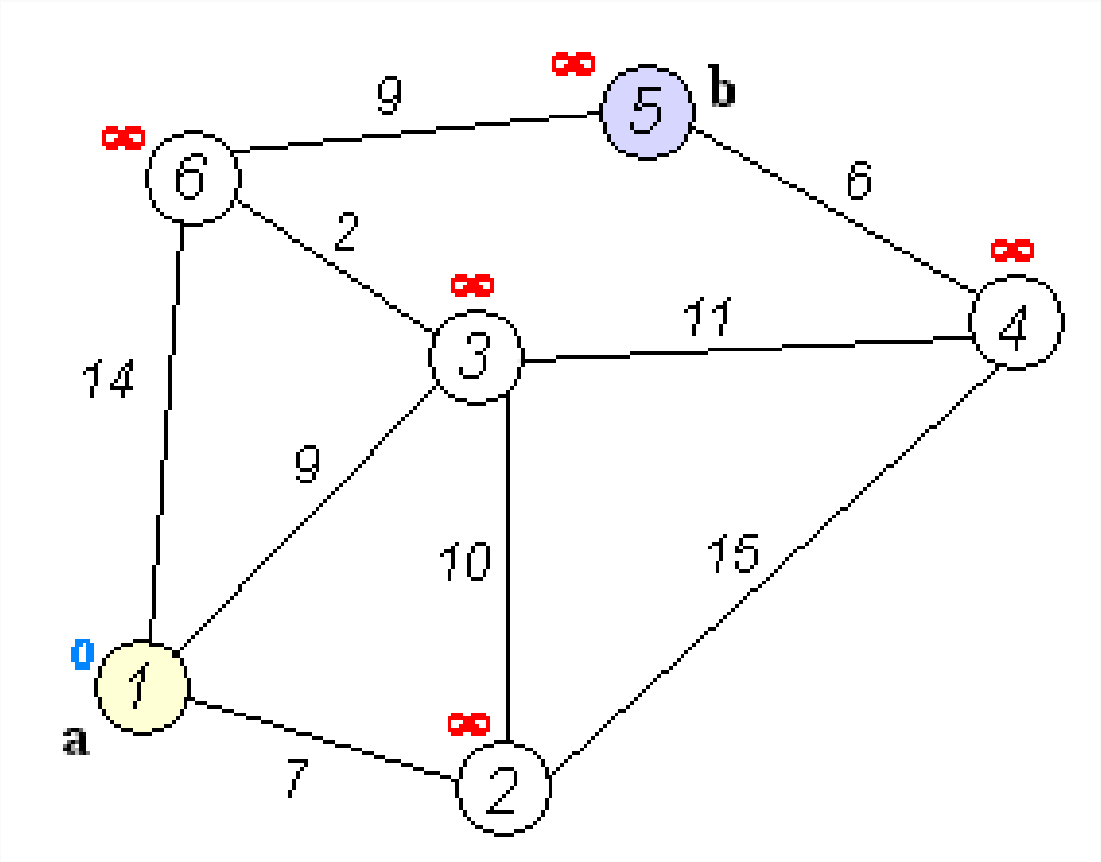
Example: Dijkstra's Shortest Path Algorithm

Algorithm:

1. Label all nodes as ∞ and "unvisited" except the origin. Start with the origin node as the current node.
2. Calculate distance of current node to all neighbors and assign the minimum of the calculated vs. labelled distance.
3. Now label the current node as "visited".
4. Select neighbor with smallest distance as new current node.

Greedily selects the shortest known path at each step

Dijkstra’s Shortest Path



Source

Divide and Conquer

Breaks down problem recursively into similar subproblems that can be solved independently, then finally combined

- This is conducive to parallelization, although repeated subproblems may be redundant
- Some cases of divide-and-conquer lead to a recursion which can introduce overhead
- Examples: sorting, median search, integer and matrix multiplication

Dynamic Programming (DP)

Breaks down problem into subproblems which are solved iteratively, where solutions are stored at each step

- DP considers all possible subproblems and combinations, and stores solutions to prevent redundancy (known as *memoization*)
- Similar to divide-and-conquer, but intermediate results are cached for later use
- Harder to implement and can require significant storage and computation time

Example: Floyd-Warshall

Algorithm (find shortest path between all node pairs):

1. Create a distance matrix where entries d_{ij} is the distance between nodes i and j. Set the distance to be:

$$d_{ij} = \begin{cases} \text{weight} & \exists \text{ edge between } i, j \\ 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

2. For all intermediate nodes k, update matrix if path through k is shorter than current direct path

```
for k in range(n):
    for i in range(n):
        for j in range(n):
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
```

Mixture of divide-and-conquer principles with dynamic programming (solving all intermediate paths and then storing it)

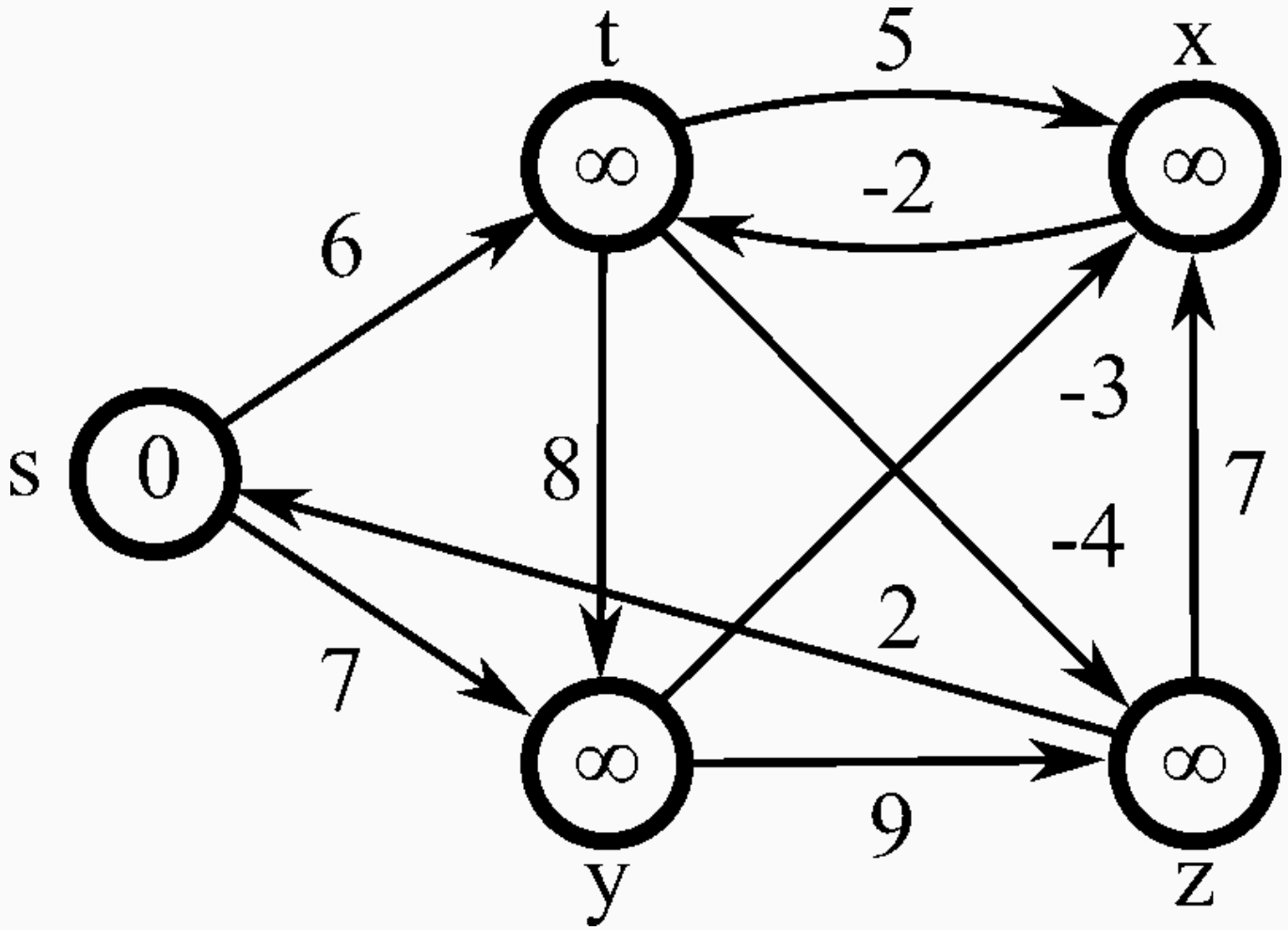
Example: Bellman-Ford

Algorithm (find shortest path between origin node and all vertices):

1. Initialize array where $d_i = \infty$ is the distance from origin node o to vertex i (except $d_o = 0$)
2. For all edges (j, i) with weight w, update $d_i = \mathbf{min}\{d_i, d_j + w\}$
3. Repeat this for all edges $N - 1$ times where N is the number of vertices (to ensure shortest path is found)
4. Repeat one last time over all edges to check there are no more updates (indicates negative cycle if so)

Iteratively relaxes edges to handle negative weights and stores all intermediate results

Bellman-Ford



Source

Combinatorial Optimization

Definition

- Optimization problems with a discrete solution set (finite or countably infinite) is combinatorial
- Theoretically possible to try all possible solutions, but solutions could be exponential (2^n) or factorial ($n!$) in number of inputs
- *Exact solution method:* find optimal solution exactly, but usually more time and storage intensive
- *Approximate solution method:* find solutions close to optimal in a tractable time frame (suited for complex/large problems)

Examples in Economics

- Matching problems
  - School choice (Abdulkadiroğlu et al., 2009)
  - Kidney exchange (Roth et al., 2004)
- Allocation problems
  - Public housing (Abdulkadiroğlu and Sonmez, 1999)
  - Job tasks (Baccara et al., 2023)
- Combinatorial decisions
  - Spectrum auctions (Cramton, 2013; Milgrom and Segal, 2020)
  - Multinational location choice (Arkolakis et al., 2023)
- Optimal network flow
  - Social networks and insurance (Karlan et al., 2009)
  - Financial network contagion (Acemoglu et al., 2015)

Techniques

- Unlike continuous optimization with a canonical solution (i.e. Newton's method), many possible combinatorial problems
  - Allocation or matching problems $\rightarrow \{0, 1\}$ outcomes
  - Network problems $\rightarrow$ set of nodes and edges
- Exact solution methods
  - Integer (linear) programming: integer decision variables
  - Branch and bound: explore branches and prune based on bounds
  - Network flow algorithms: adaptable to assignment problems
- Approximation solution methods
  - Local search: incrementally move to better solution (greedy)
  - Simulated annealing: probabilistic acceptance of worse solution

Linear Programming (LP)

Objective and constraints are all linear (canonical form):

$$\max_{x \in \mathbb{R}^n} \quad C^T x \quad s.t. \quad Ax \leq b, \quad x \geq 0$$

- Linear programs are a special case of *continuous* optimization problems but solution set is *finite*
- Simplex algorithm moves along edges of feasible region (constructed by constraints) to find optimal vertex
- Simplex algorithm is exponential time in worst-case scenario, but other solution methods are polynomial time
  - Interior point methods, ellipsoid algorithms

Integer Linear Programming (ILP)

Objective and constraints are all linear, and solution belongs to set of integers $\mathbb{Z}$ (canonical form):

$$\max_{x \in \mathbb{Z}^n} \quad C^T x \quad s.t. \quad Ax \leq b, \quad x \geq 0$$

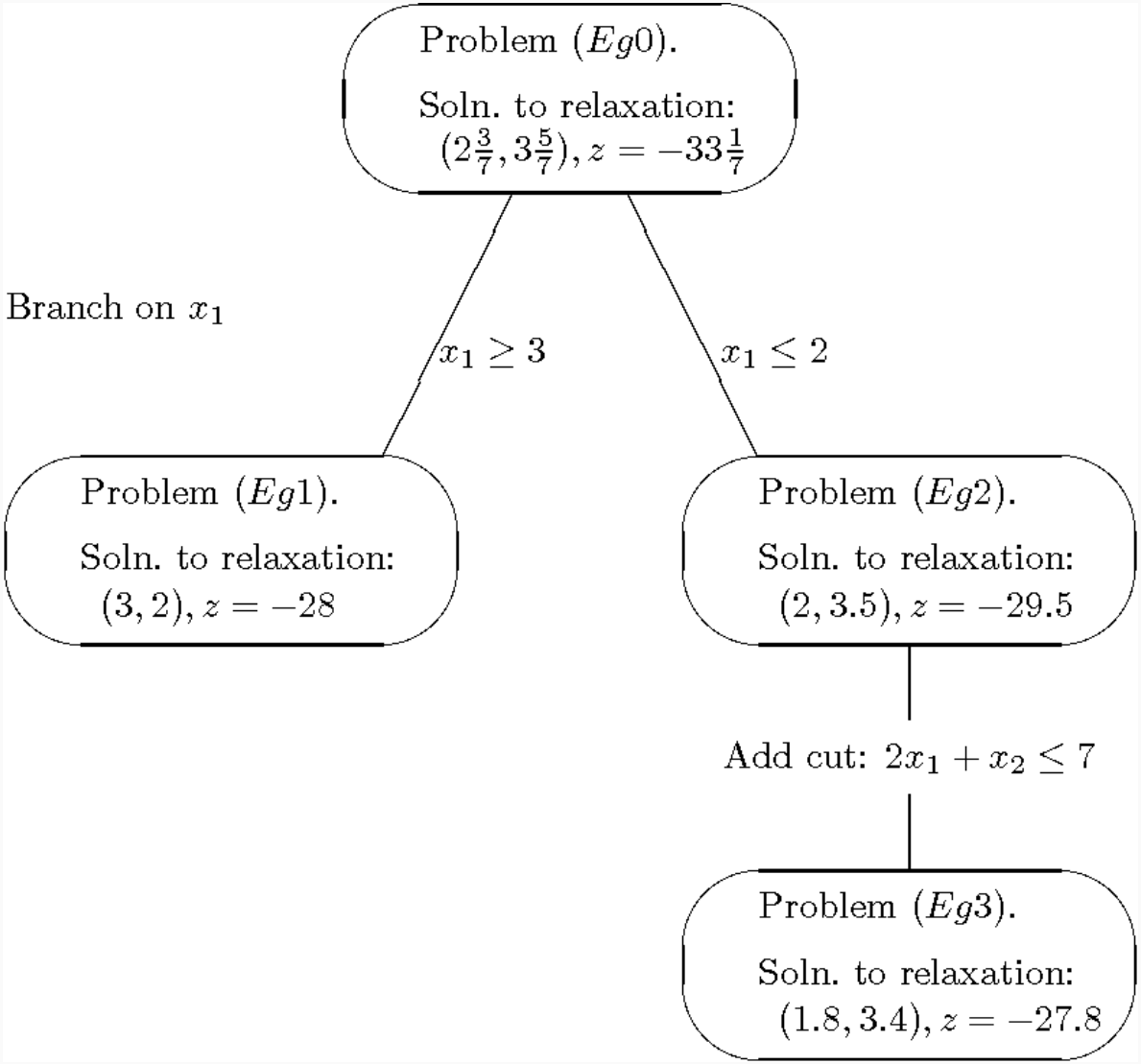
- Unlike LP, ILP is NP-hard but we can use LP
- Idea: combine branch and bound with cutting planes $\rightarrow$ branch and cut
  - Algorithms to generate additional constraints to "cut off" a vertex
- Applicable to a variety of problems:
  - $x_i \in \{0, 1\}$ on whether to invest in project/locate new stores
  - $x_{ij} \in \{0, 1\}$ on matches between i and j subject to constraints
  - $x_i \in \mathbb{Z}$ is number of discrete item i

Branch and Cut

1. Solve the LP relaxation of the ILP problem to get x^*
   - A, b have integer entries and every square submatrix has $det \in \{-1, 0, 1\} \Rightarrow$ every solution is integral
2. Form branches for fractional values x_i^* via two new subproblems: (1) $x_i \leq \lfloor x_i^* \rfloor$ and (2) $x_i \geq \lceil x_i^* \rceil$
3. Solve LP relaxation for subproblems and keep track of upper and lower bound of solution
4. For each node (including original), add cutting planes (e.g. Gormory's algorithm) and solve LP relaxation
5. Prune subproblems/branches which are infeasible or worse than the current best integer solution
6. Repeat branching, cutting, and pruning until all branches are pruned or integrally solved

Branch and Cut

In this case, we are minimizing an objective function:

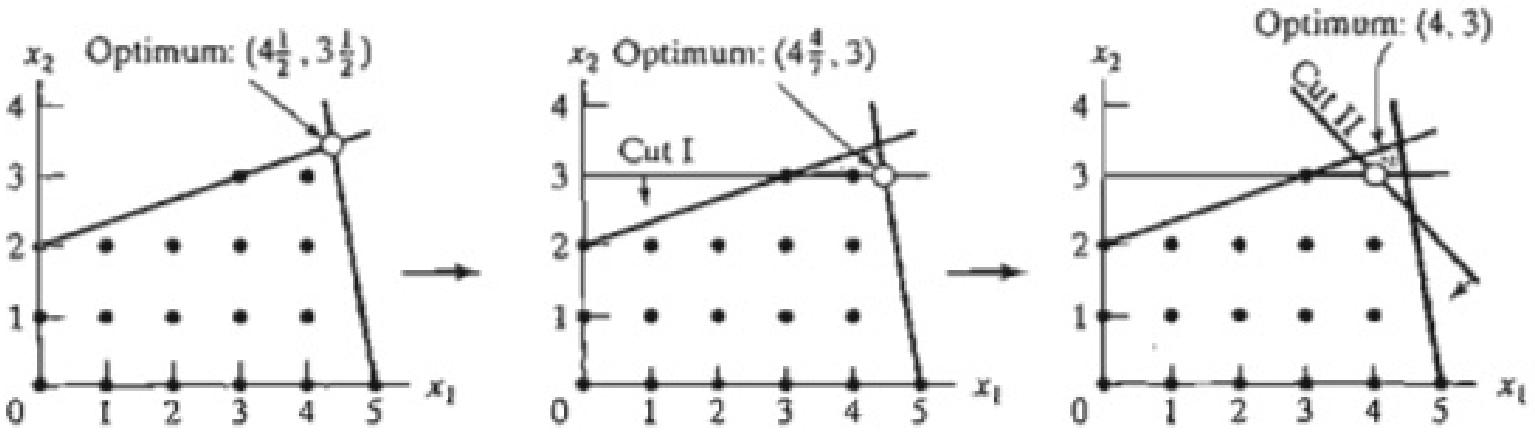


Source

Cutting Plane Algorithm

FIGURE 9.10

Illustration of the use of cuts in ILP



Source

Simulated Annealing

Idea: Explore local solutions and accept worse solutions with some probability that decreases with time

- Comes from metallurgy in which the metal cools down with time
- Adaptation of Metropolis-Hastings
- S is solution and T is "temperature"
- Objective function: $f(S)$
- Acceptance probability: $P(f(S), f(S'), T)$
- Cooling schedule: $c(T)$

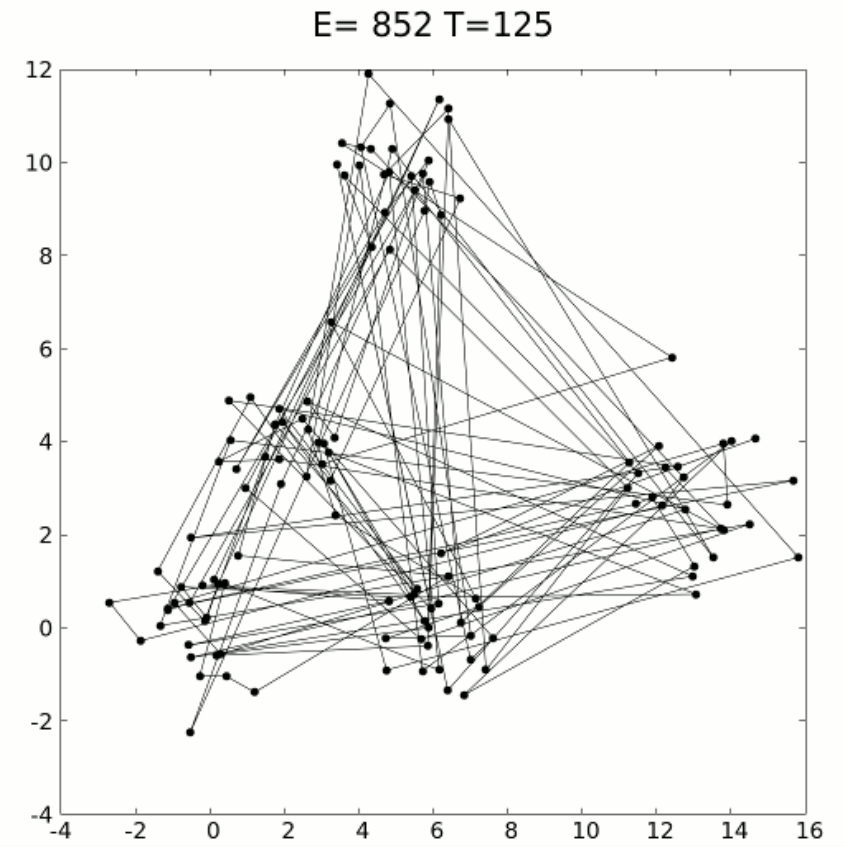
Simulated Annealing Algorithm

Suppose we are minimizing the objective function:

1. Generate neighboring solution of S, call it S'
   - E.g., Permutation of any two nodes to be visited
   - E.g., Increase/decrease solution by 1 in ILP
2. If $f(S') < f(S)$, accept S' as new S. If not, accept S' with probability $P(f(S), f(S'), T)$
   - Common probability is $\mathbf{exp}[-(f(S') - f(S))/T]$
3. Reduce temperature according to cooling schedule $c(T)$
   - E.g., $T \leftarrow \alpha T, \alpha \in (0, 1)$
   - E.g., $T \leftarrow \frac{1-(k+1)}{K}$ for max iterations K
4. Return final solution when T reaches threshold $\epsilon > 0$

Example: Traveling Salesman Problem

Find the shortest possible route visiting a set of cities exactly once and returns to starting city (given cities and distance pairs) – this is NP-hard!

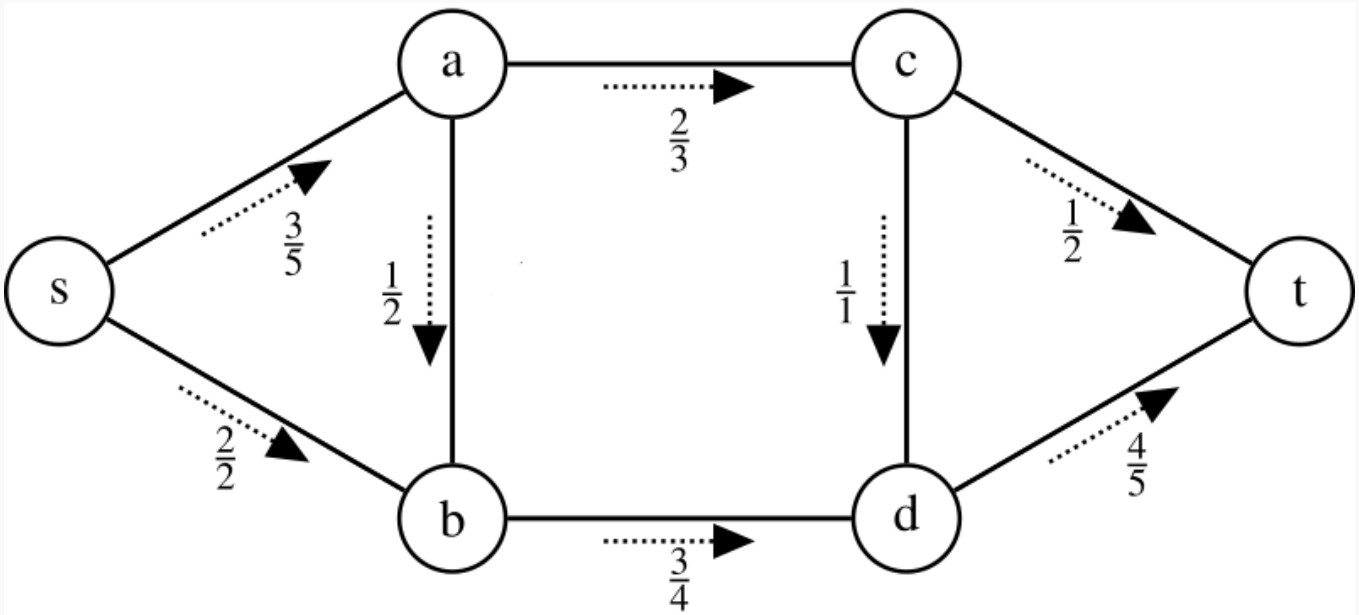


Source

Network Flow

A flow network consists of:

- Graph $G = (V, E)$ with vertices V and *directed* edges E
- $s, t \in V$ where s is the source and t is the sink
- Capacity $c(e) \quad \forall \quad e \in E$



Source

Minimum Cut and Maximum Flow

What is the maximum flow of the network which satisfies the following properties:

- Capacity: $0 \leq f(e) \leq c(e)$ for all $e \in E$
- Flow conservation: net inflow $=$ net outflow for all $v \in V/\{s, t\}$

Define a cut as a partition (A, B) of nodes V s.t. $s \in A, t \in B$

- The capacity of a cut is the $\sum_e' c(e')$ for all e' going from A to B
- What is the minimum capacity of a cut?

Theorem
Max-Flow Min-Cut: The maximum flow value is equal to the minimum capacity over all cuts. If a cut value and a flow value are found to be equal, they are the min cut and max flow.

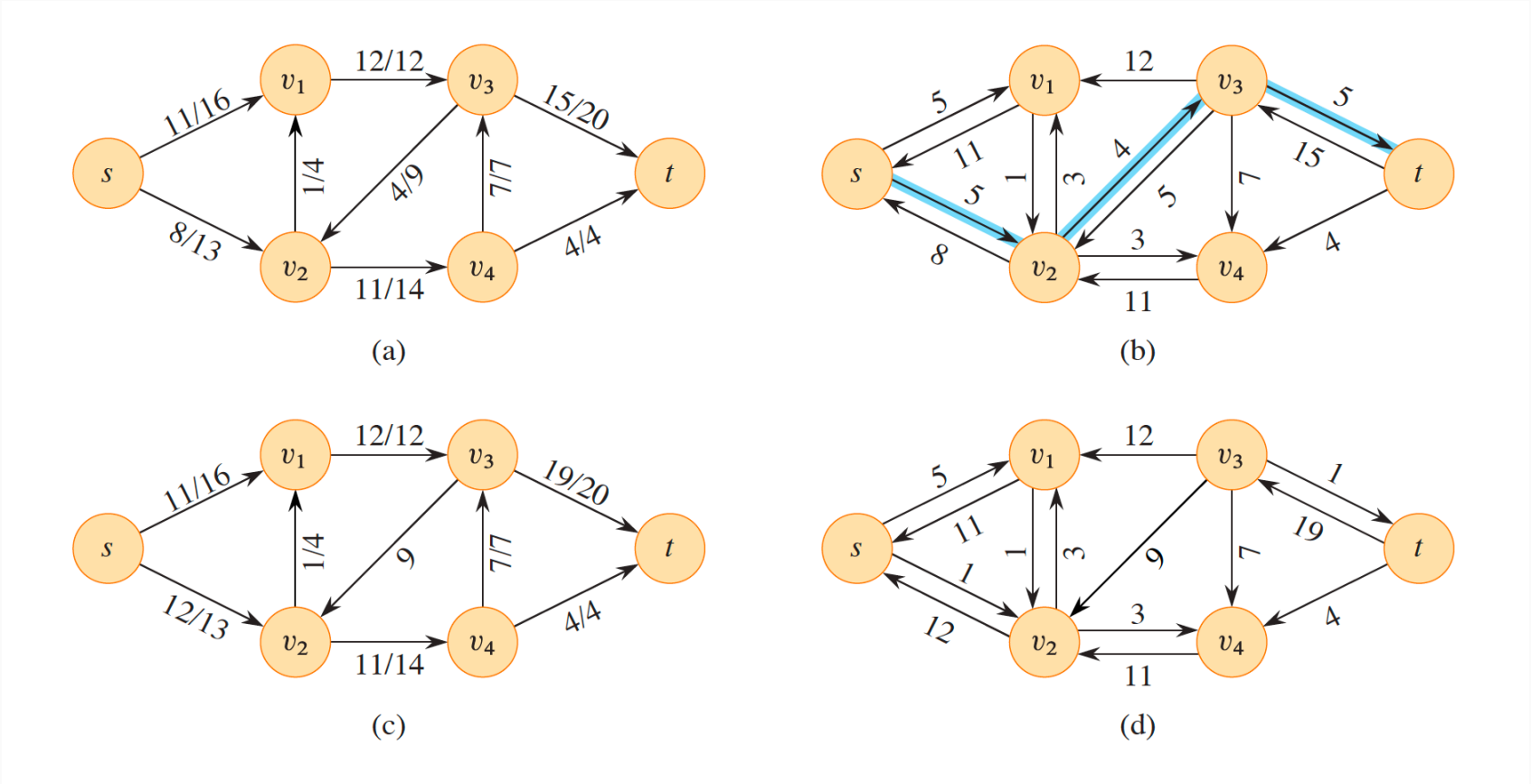
Ford-Fulkerson Algorithm

How do we find the maximum flow?

1. Assign all flow of edge $u \to v$ as $f(u, v) = 0$
2. Create residual graph G_f which has residual capacity $c(e) - f(e) = 0$ along the edges
3. Find one possible path from source to sink on G_f via tree search (breadth-first or depth-first)
4. Take the minimum residual capacity along this augmented path and add that flow to $f(e)$
5. Update the residual graph and repeat until no more augmented paths exist

With breadth-first search, complexity is $O(|V||E|^2)$

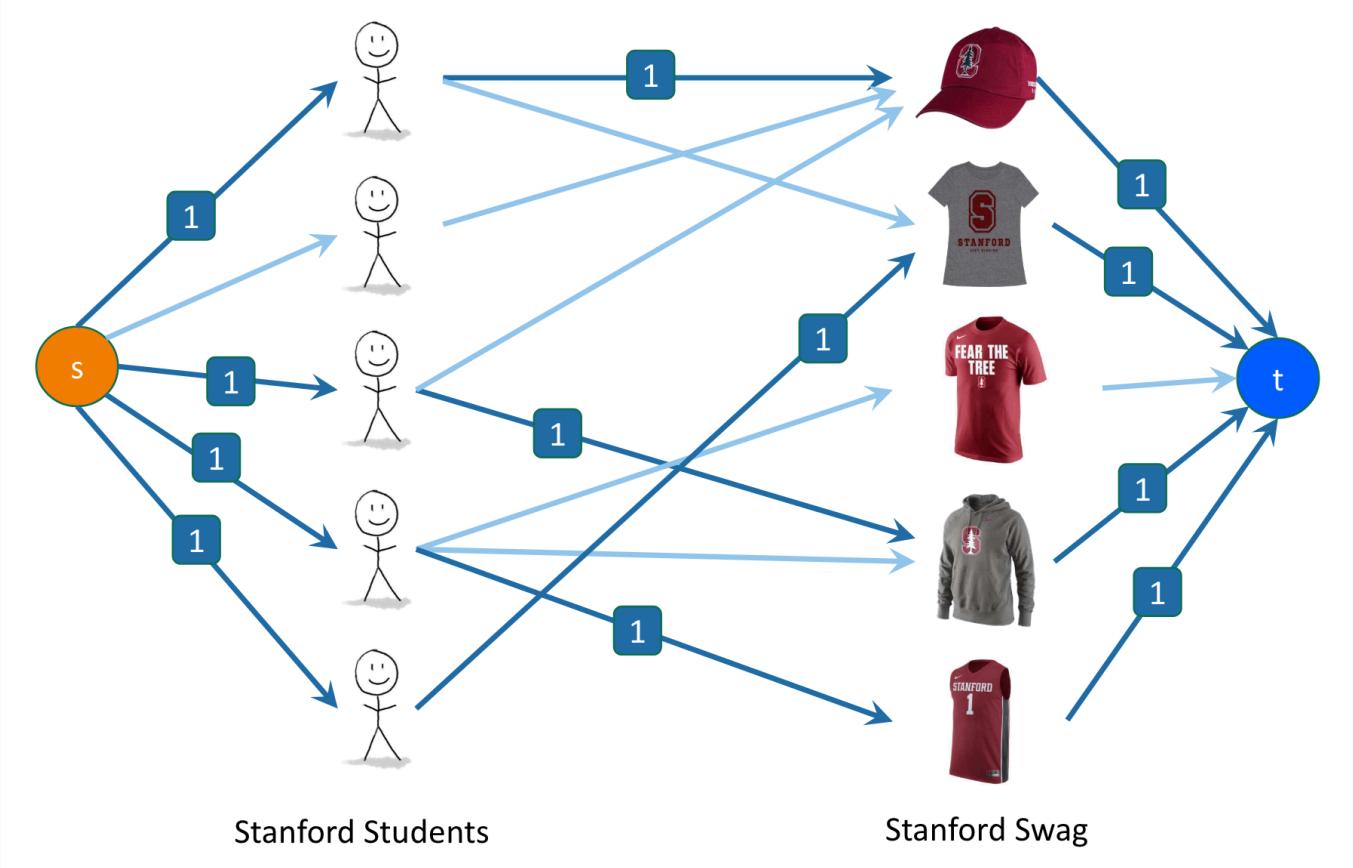
Residual Capacity



Source: “Introduction to Algorithms” page 678

Network Flow and Assignment Problems

Classic assignment problem is a maximum flow problem with a bipartite graph and edge capacities of one



Source