Finals revision notes

Taxonomy of network problems (and methods)

- **Linear programming (Simplex)**
  - **Min-cost flow problem (Network Simplex)**

    For our course we will assume all edges have infinite capacity.

    - **Max-flow problem (Ford-Fulkerson)**

      This can be formulated into a min-cost flow problem by connecting an arc from the tap of the source, with infinite capacity and negative one cost.

      (The dual problem is the **min-cut problem**.)

      - Baseball elimination (example)

        (Only have four layers, attempts to exhaust all supply)

    - Maximum cardinality matching
    - Transhipment problem

      How to meet the demand while minimising delivery costs?

      Can be formulated into a min-cost flow problem by adding dummy sources or taps with zero cost arcs. This makes the total demand equals to the total supply.

    - **Shortest path problem (Dijkstra)**

      You are transporting a unit supply from the source to the tap, at minimum cost

    - Transportation problem

      Transhipment problem without intermediate nodes.

      - Assignment problem

        Transhipment problem but the total demand is equal to the total supply.

The max-flow problem

Given a directed network $G = (V, E)$ and a source node s, a sink node t, and a possibly infinite capacity u_{ij} for each arc (i, j)

The max-flow problem is the problem of finding a value of flow for every arc in such a way that

- Flows are nonnegative and arc capacities are satisified
- For every node that is not s or t, the incoming flow is equal to the outgoing flow
- The flow entering the sink t is maximum

Formulation of the Linear Program

$$
\begin{aligned}
&\text{Objective function} \\
&\max q \\
&\text{The inflow of the source} \\
& && \sum_{(s,j)\in E} x_{sj} && = q \\
&\text{The outflow of the tap} \\
& && \sum_{(i,t)\in E} x_{it} && = q \\
&\text{For all other nodes, inflow equals outflow} \\
&\forall v \in V, v \neq s, t && \sum_{(v,j)\in E} x_{vj} - \sum_{(i,v)\in E} x_{iv} && = 0 \\
&\text{Capacity constraint} \\
&\forall (i,j) \in E && x_{ij} && \leq u_{ij} \\
&\text{Nonnegativity constraint} \\
&\forall (i,j) \in E && x_{ij} && \geq 0
\end{aligned}
$$

Transforming into a max-flow problem

- Edge with infinite flow - use a very large number as capacity for computation.
- Multiple source - create a single dummy source that has a directed edge to each of the original source. The capacity of the edge is the capacity of the original source. The original source is now like any other node.
- Multiple taps - create a single dummy tap that has a directed node from each of the original tap. The capacity of the edge is the capacity of the original tap. The original tap is now like any other node.

Ford-Fulkerson algorithm for max-flow

Problem parameters

These are the numbers that you need to keep track of

- **Nodes**
    - Provided by the problem
        - **Label**. This is how the node is referred to. Be If this is also a number it can be quite confusing.
        - (There are no require demand or supply)
- **Edges** (directed)
    - Provided by the problem
        - Capacity of the edge
            - You can consider that between every pair of node there is an edge, we do not show one with zero capacity.
    - Calculated for each iterations
        - Modified capacity
            - If the capacity is initially zero, we augment the arc.

Initialise the **current flow value** as zero

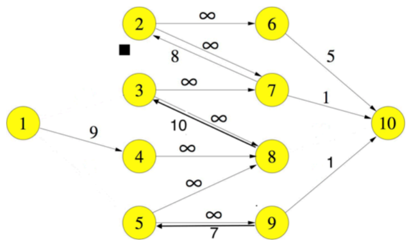
Iterate with the following steps

- Find any **directed path** P from the source to the tap.
- Send the **maximum possible flow** f along P.
- **Add** f to the current flow value.
- **Update the path** P.
    - Decrease the forward capacity by f.
    - Increase the backward capacity by f
        - You may need to create an (augmenting) path.
    - If a directed edge from the source is zero, remove.
    - If a directed edge to the tap is zero, remove.

Terminate the algorithm when there are no more directed paths from the source to the tap. The complexity of the algorithm depends on the choice of augmenting paths.

The backflow capacity needs to be decreased if used (from 10 to 9 in the example below).

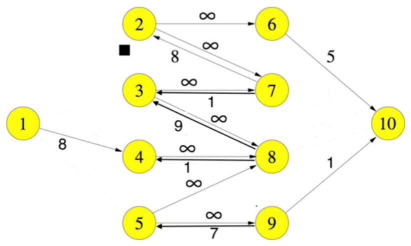
Step 2: Send 7 units of flow along $1 \to 5 \to 9 \to 10$:



Current flow value is 25.

example of residual network

Step 3: Send 1 unit of flow along $1 \to 4 \to 8 \to 3 \to 7 \to 10$:



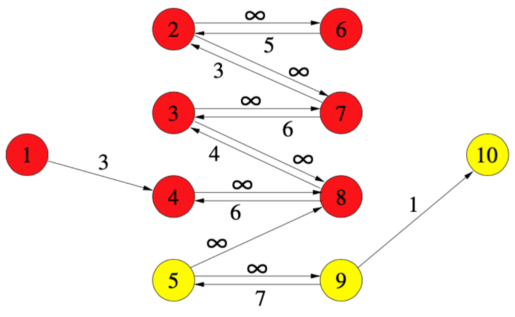
Current flow value is 26.

My understanding of the intuition

- when you assign the flow you also assign **flexibility** with the backflow values
- **progress** is made when the edges from the source and tap is removed, eventually leaving no directed node from the source to the tap

Finding the min-cut from the max-flow

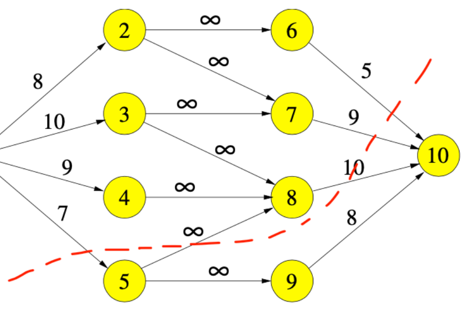
From the final and optimal residual network, find all nodes that can be reached from s. These nodes form the S side of the cut, all other nodes are on the T side.



What is the value of the cut?

Finding the min-cut from the max-flow

Let $S = \{1, 2, 3, 4, 6, 7, 8\}$ and $T = \{5, 9, 10\}$, then the value of th cut is 31.



This is the min cut!

The backflow is ignored in the calculation of the value of the min-cut.

Each feasible solution of the dual problem provides an upper bound to the primal problem.

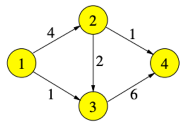
Natuaral bounds - if you partition right outside the source and the tap, you get trivial upper bounds. It is not possible to send any more than the source can send, or any more than the sink can take.

The optimal solutions for the primal and dual problems is the same.

Primal-dual pair for max-flow problem

The dual of max-flow problem is the min cut problem. Assigning the d-values provides a partition.

Consider the following network, write the LP formulation and take its dual.



The LP formulation for the max-flow:

$$
\left.\begin{array}{lrcl}
\max & q & & \\
 & x_{12}+x_{13}-q & = & 0 \\
 & -x_{12}+x_{23}+x_{24} & = & 0 \\
 & -x_{13}-x_{23}+x_{34} & = & 0 \\
 & -x_{24}-x_{34}+q & = & 0 \\
 & x_{12} & \leqslant & 4 \\
 & x_{13} & \leqslant & 1 \\
 & x_{23} & \leqslant & 2 \\
 & x_{24} & \leqslant & 1 \\
 & x_{34} & \leqslant & 6 \\
\forall(i,j) & x_{ij} & \geqslant & 0 \\
 & q & & \text{free}
\end{array}\right\}
$$

Wikipedia offers good explanation https://en.wikipedia.org/wiki/Max-flow_min-cut_theorem
Symbols differ.

Take the dual:

$$
\left.\begin{array}{lrcl}
\min & 4y_{12}+1y_{13}+2y_{23}+1y_{24}+6y_{34} & & \\
 & d_1-d_2+y_{12} & \geqslant & 0 \\
 & d_1-d_3+y_{13} & \geqslant & 0 \\
 & d_2-d_3+y_{23} & \geqslant & 0 \\
 & d_2-d_4+y_{24} & \geqslant & 0 \\
 & d_3-d_4+y_{34} & \geqslant & 0 \\
 & -d_1+d_4 & = & 1 \\
\forall(i,j) & y_{ij} & \geqslant & 0 \\
 & d_1,d_2,d_3,d_4 & & \text{free.}
\end{array}\right\}
$$

	Max-flow (Primal)	**Min-cut (Dual)**
variables	$f_{uv}\ \forall(u,v)\in E$ *[a variable per edge]*	$d_{uv}\ \forall(u,v)\in E$ *[a variable per edge]* $z_v\ \forall v\in V\setminus\{s,t\}$ *[a variable per non-terminal node]*
objective	maximize $\sum_{v:(s,v)\in E} f_{sv}$ *[max total flow from source]*	minimize $\sum_{(u,v)\in E} c_{uv}d_{uv}$ *[min total capacity of edges in cut]*
constraints	subject to $\begin{array}{rcll} f_{uv} & \leq & c_{uv} & \forall(u,v)\in E \\ \sum_u f_{uv}-\sum_w f_{vw} & = & 0 & v\in V\setminus\{s,t\} \end{array}$ *[a constraint per edge and a constraint per non-terminal node]*	subject to $\begin{array}{rcll} d_{uv}-z_u+z_v & \geq & 0 & \forall(u,v)\in E, u\neq s, v\neq t \\ d_{sv}+z_v & \geq & 1 & \forall(s,v)\in E \\ d_{ut}-z_u & \geq & 0 & \forall(u,t)\in E \end{array}$ *[a constraint per edge]*
sign constraints	$f_{uv}\ \geq\ 0\quad\forall(u,v)\in E$	$\begin{array}{rcll} d_{uv} & \geq & 0 & \forall(u,v)\in E \\ z_v & \in & \mathbb{R} & \forall v\in V\setminus\{s,t\} \end{array}$

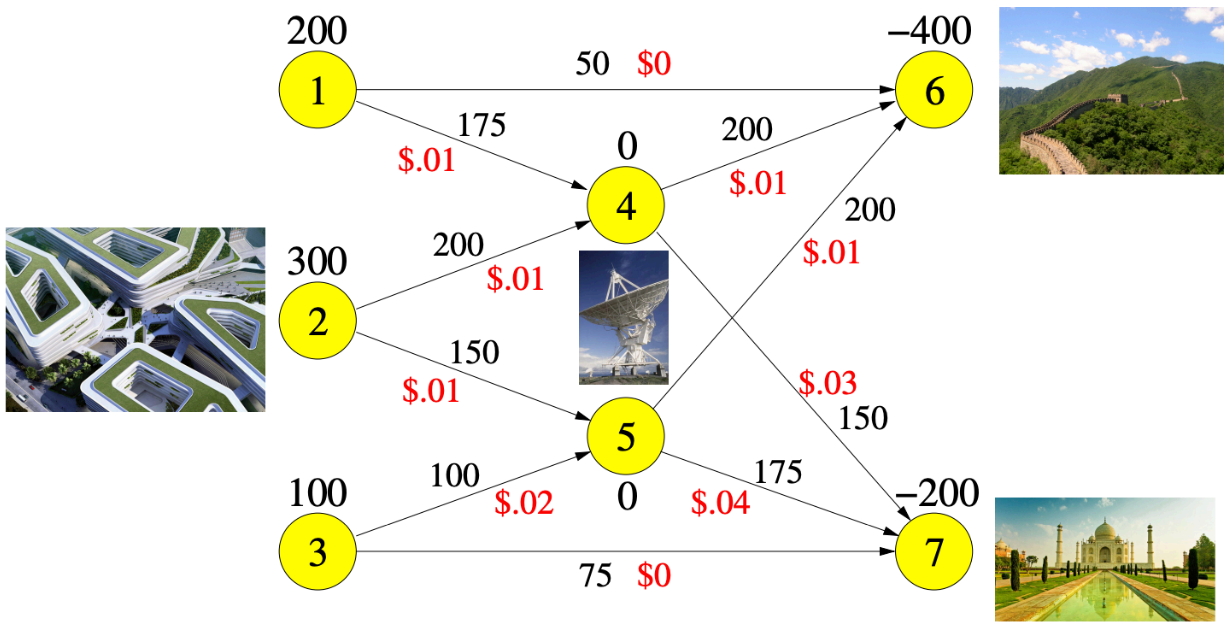
Min-cost flow problem

This is a more general form of a max-flow problem, but still a specific class of linear programming problem (refer to taxonomy)

Given a network $G = (V, E)$ with possibly infinite arc capacities u_{ij}, arc costs c_{ij}, and a supply/demand b_i for each node, the corresponding minimum cost flow problem is the following:

$$\left.\begin{array}{rrcl} \min & \sum_{(i,j)\in E} c_{ij}x_{ij} & & \\ \forall i \in V & \sum_{(i,j)\in E} x_{ij} - \sum_{(j,i)\in E} x_{ji} & = & b_i \\ \forall (i,j) \in E & x_{ij} & \leqslant & u_{ij} \\ & x_{ij} & \geqslant & 0. \end{array}\right\}$$

For each node the outflow minus inflow equal b_i. If $b_i > 0$, node i is a supply node, if b_i is a demand node, node i is a demand node.



$$\left.\begin{array}{rr}
\min & 0(x_{16}+x_{37})+0.01(x_{14}+x_{24}+x_{25}+x_{46} \\
 & +x_{56})+0.02x_{35}+0.03x_{47}+0.04x_{57} \\
\text{(Flow node 1)} & x_{14}+x_{16}=200 \\
\text{(Flow node 2)} & x_{24}+x_{25}=300 \\
\text{(Flow node 3)} & x_{35}+x_{37}=100 \\
\text{(Flow node 4)} & x_{46}+x_{47}-x_{14}-x_{24}=0 \\
\text{(Flow node 5)} & x_{56}+x_{57}-x_{25}-x_{35}=0 \\
\text{(Flow node 6)} & -x_{16}-x_{46}-x_{56}=-400 \\
\text{(Flow node 7)} & -x_{37}-x_{47}-x_{57}=-200 \\
 & x_{16}\leqslant 50, x_{14}\leqslant 175 \\
 & x_{24}\leqslant 200, x_{25}\leqslant 150 \\
\text{(Capacity constraints)} & x_{35}\leqslant 100, x_{37}\leqslant 75 \\
 & x_{46}\leqslant 200, x_{56}\leqslant 200 \\
 & x_{47}\leqslant 150, x_{57}\leqslant 175 \\
\forall(i,j) & x_{ij}\geqslant 0.
\end{array}\right\}$$

Properties

- Feasible solution is not guaranteed - it is possible that the supply do not reach the demand.
- Redundancy - the last flow node is a linear combination of rest of the flow nodes.

Transforming into a min-cost problem In the min-cost problem total demand should be equal to the total supply $\sum_i b_i = 0$, or else there this is not a min-cost problem (or no feasible solution). Therefore

- If the supply exceeds the demand, create a dummy sink connected to each source, each with **zero cost**.
- If the demand exceeds the supply, create a dummy source connected to each source, each with **very high cost**.

Converting a max-flow problem into a min-cost problem

Every max-flow problem can be converted into a min-cost problem (refer to taxonomy of linear programming problems).

Add an arc from the sink to the source with infinite capacity. The objective function is to minimise the negative of the flow of the new sink-source arc. The cost of the rest of the arcs are zero.

Network simplex

A tree is a graph **with no cycles**. It can be directed or undirected, but we consider undirected trees.

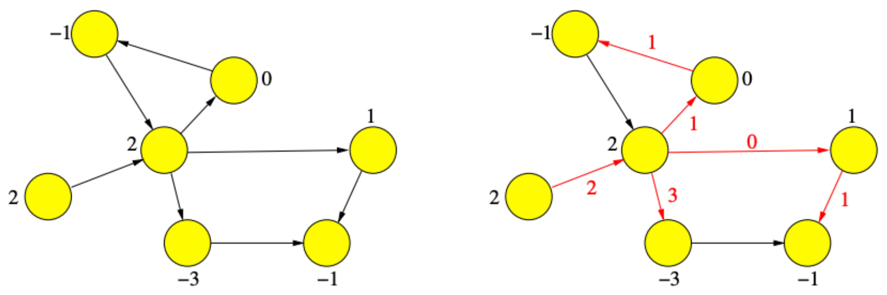
A spanning tree of a connected graph is a subset of its arcs that forms an undirected tree touching every node. (Directions does not matter here)

The basic feasible solution of the LP formulation of a min-cost flow problem correspond to a spanning tree.

The **basic feasible solution** is a spanning tree that allows all supplies or demands to be satisfied.

Bfs for min-cost flow

The basic feasible solutions of the LP formulation of a min-cost flow problem correspond to a spanning tree.



The basic feasible solution is a spanning tree that allows all supplies/demands to be satisfied. Note that the spanning trees consider arcs as undirected, but the flows must agree with arc directions.

LP formulation of a min-cost flow problem

Given a network $G = (V, E)$ with possibly infinite arc capacities u_{ij}, arc costs c_{ij}, and a supply/demand b_i for each node, the corresponding minimum cost flow problem is the following:

$$\left.\begin{array}{rrcl}
\min & \sum_{(i,j)\in E} c_{ij}x_{ij} & & \\
\forall i \in V & \sum_{(i,j)\in E} x_{ij} - \sum_{(j,i)\in E} x_{ji} & = & b_i \\
\forall (i,j) \in E & x_{ij} & \leqslant & u_{ij} \\
 & x_{ij} & \geqslant & 0.
\end{array}\right\}$$

The capacity limits u_{ij} will not be considered in this course.

Problem parameters

These are the numbers that you need to keep track of

- **Nodes**
    - Provided by the problem
        - **Label**. This is how the node is referred to. If this is also a number it can be quite confusing.
        - **Supply** (positive) or **demand** (negative)
            - The total demand must be equal to the total supply.
    - Calcuated for each BFS
        - **Simplex multiplier** y_i
            - Down the directed edge of the basic arc, the simplex multiplier decreases by the initial cost of the edge.
- **Edges** (directed)
    - Provided by the problem
        - Capacity (ignored, this course assumes infinite capacity)
        - **Initial cost** c_{ij}
    - Calculated for each BFS
        - **Assigned flow**
            - If the current flow is zero, it is a nonbasic arc
            - If the current flow is **positive**, it is a **basic** arc
            - Flow cannot be negative
        - **Reduced cost** $\bar{c}_{ij}$
            - The reduced cost of every nonbasic arc is calculated
                - (The reduced cost of every basic arc is zero)
            - $\bar{c}_{ij} = c_{ij} - y_i + y_j$
            - Initial cost minus decrease in simplex multiplier down the direction of the edge

Algorithm

Initialising the network simplex

You start with a **basic feasible solution** a minimum spanning tree that allows all supplies and demands to be satisfied.

Compute the **simplex multipliers** starting from a leaf (chosen at random or instructed). The leaf (which is a node) is set to zero.

Recurse until solution is optimal

Compute the **reduced costs**. If all reduced cost are **nonnegative**, the solution is optimal.

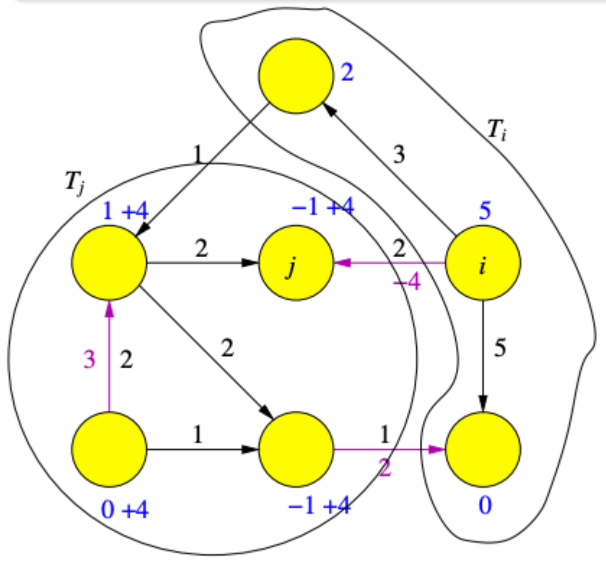
Pivot to improve the BFS. "We try to send as much flow as possible along that arc, while staying feasible". In other words, you **redirect** the flow to the nonbasic arc, from the (series of) basic arcs, until one of the flow of the basic arc is zero.

Update the simplex multipliers. You can repeat the same process, or use the following faster procedure. Then, repeat the cycle.

Fast update procedure for simplex multipliers

Call (i,j) the arc entering the basis. After the pivot there are two trees: T_i that contains i, T_j that contains j.

- Leave the simplex multipliers of nodes in T_i untouched.
- Increase the simplex multipliers of nodes in T_j by $|\bar{c}_{ij}|$.

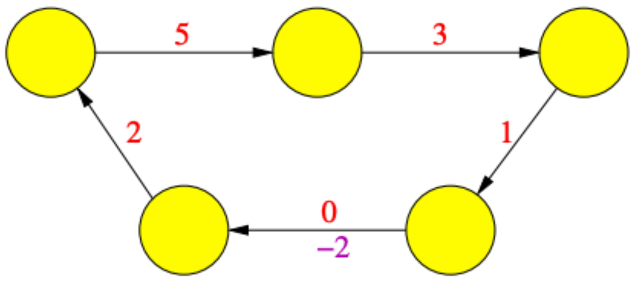


This way, all simplex multipliers are recomputed very quickly performing a small number of additions.

Assumptions

- No capacity constraints for the edges (the exam will not have constrains for edge capacity) How it is modified - set the nonbasic variables to either the upper or lower bound. Pivot on this edge only if the reduced cost is negative and if the nonbasic variable is at the lower bound, or the reduced cost is positive if the nonbasic variable is at the upper bound.
- The problem is feasible - it is already if you can find a basic feasible solution.
- The problem is not unbounded. If you find a cycle that has a negative total cost along the path, the cost can be minimised indefinitely.

When determining the flow of an arc entering the basis, we may find that all arcs in the cycle are in the same direction, as below (reduced cost for the nonbasic arc in purple, flows in red):



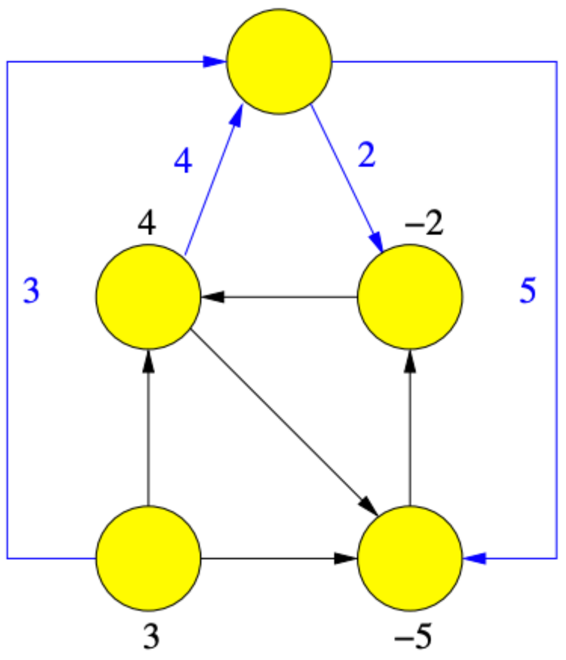
In this case, using the t method shows that we can increase the flow on the arcs as much as we want. Because the nonbasic arc has negative reduced cost, it means that every unit of flow on this cycle decreases the cost.

The problem is unbounded: the cost can go to $-\infty$.

Phase I of network simplex

Do this to obtain a basic feasible solution for the network simplex.

We can think of adding artificial arcs from an artificial node that receives all the supply and redistributes it to satisfy all the demand.



The initial flow on the artificial arcs is simply the supply/demand from the corresponding node in the original network.

Objective: minimize the sum of the flow on the artificial arcs.

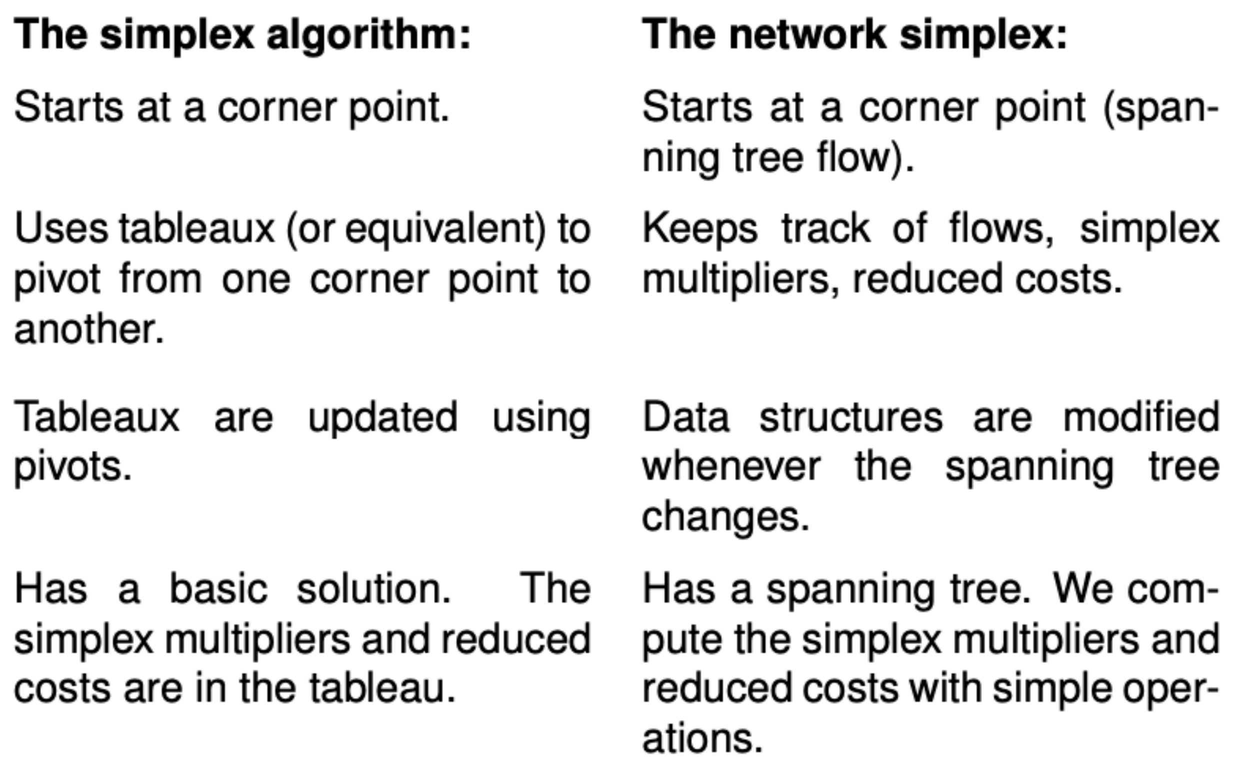
(Following comment needs to be confirmed, I am not too sure.)

The flow on the actual arcs is zero, and the flow on the artificial arcs is equal to the demand or supply of the node.

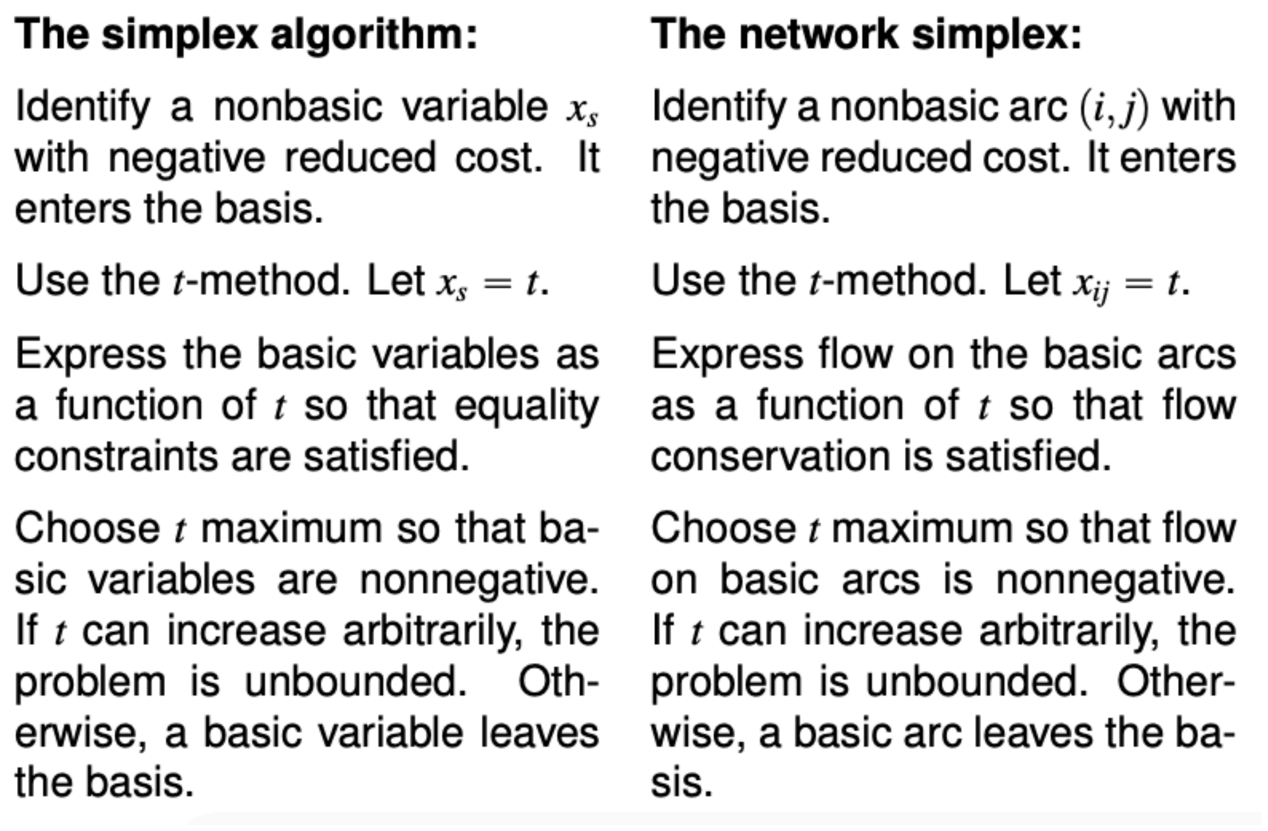
To obtain a basic feasible solution, we want to redirect the flow on the artificial arcs to the actual arcs. We set the costs on the artificial arcs to one, and the costs on the actual arcs to zero.

If you cannot reduce the cost on this modified problem to zero, you cannot obtain a minimum spanning tree from the actual arcs, and there is no feasible solution.

Comparison between simplex and network simplex



The simplex algorithm:	The network simplex:
Starts at a corner point.	Starts at a corner point (spanning tree flow).
Uses tableaux (or equivalent) to pivot from one corner point to another.	Keeps track of flows, simplex multipliers, reduced costs.
Tableaux are updated using pivots.	Data structures are modified whenever the spanning tree changes.
Has a basic solution. The simplex multipliers and reduced costs are in the tableau.	Has a spanning tree. We compute the simplex multipliers and reduced costs with simple operations.



The simplex algorithm:	The network simplex:
Identify a nonbasic variable x_s with negative reduced cost. It enters the basis.	Identify a nonbasic arc (i, j) with negative reduced cost. It enters the basis.
Use the t-method. Let $x_s = t$.	Use the t-method. Let $x_{ij} = t$.
Express the basic variables as a function of t so that equality constraints are satisfied.	Express flow on the basic arcs as a function of t so that flow conservation is satisfied.
Choose t maximum so that basic variables are nonnegative. If t can increase arbitrarily, the problem is unbounded. Otherwise, a basic variable leaves the basis.	Choose t maximum so that flow on basic arcs is nonnegative. If t can increase arbitrarily, the problem is unbounded. Otherwise, a basic arc leaves the basis.

Integer programming

Memorising the solutions to each problem does not scale. You need to understand the design priniciple behind every solution.

Example - flipping the board of five. Takeaways

- How to constrain a linear expression to **odd and even** (use $2k + 1$, k integer)
- Use of dummy variables outside the grid to **simplify expressions**
- **Minimising an integer variable** also constrains the variable to take only **binary** values.

Example - IKEA with setup costs. Takeaways

- Use of a **big-M** constraints to add "**setup cost**" variable and constraint.

  $x \leq My$

  - when y is zero, x is constrained to be **zero**
  - when y is one, no constraints is placed on x

Modelling ILP problem

Linear programming can only solve convex problems, with inclusive boundaries, and you cannot multiply variables.

Strategy - transform the problem into ORs and ANDs.

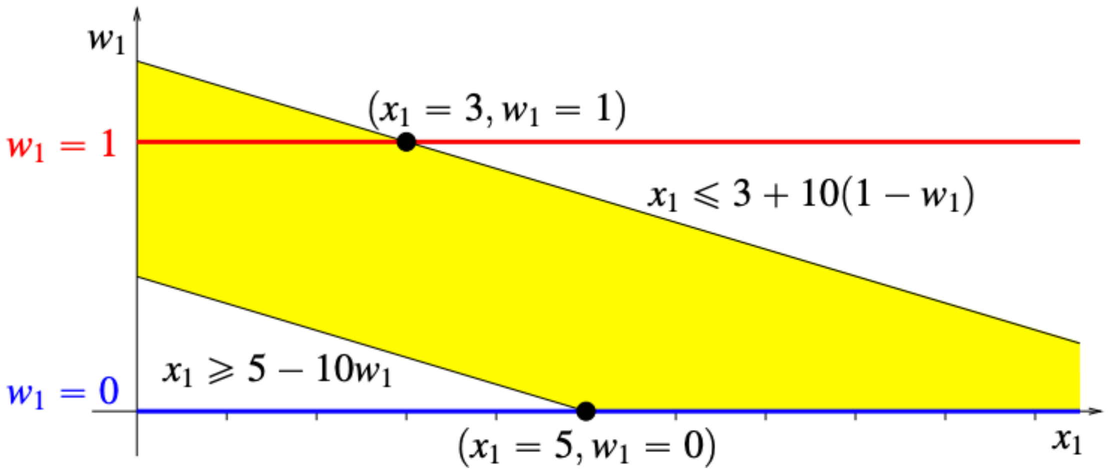
Modelling less-than-or-greater-than (non-convex) inequality

$$x \leq 2 \quad \text{or} \quad x \geq 6$$

You can turn off either constraint by adding a binary variable w_i with a big M.

$$x \leq 2 + Mw \quad \text{and} \quad x \geq 6 - M(1-w) \quad \text{and} \quad w \text{ binary}$$

Consider the geometry of the formulation.

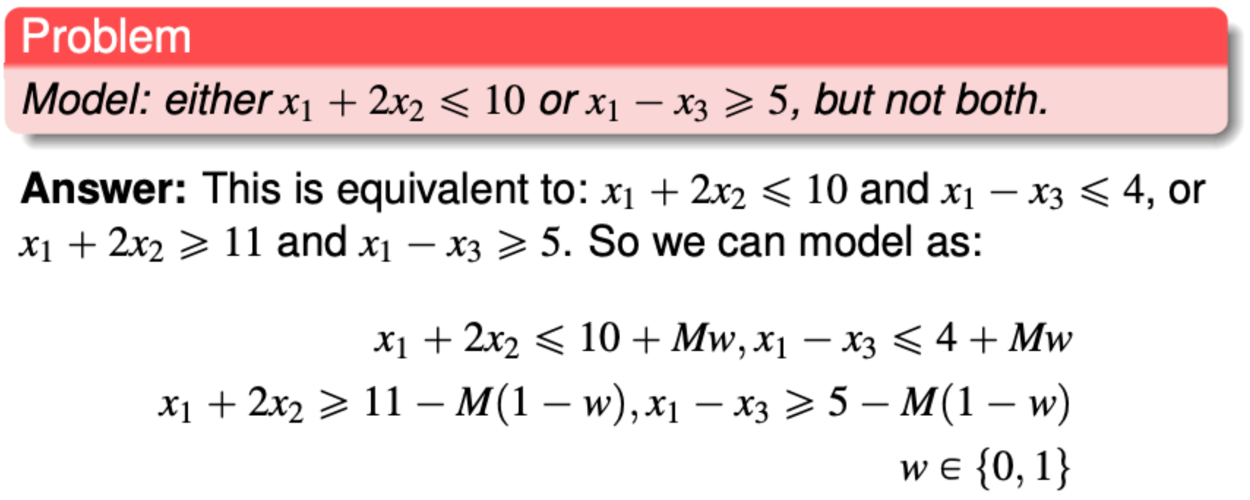


The yellow region is convex.

Modelling with complement constraint

One and only one of the constraint and its complement can be fulfilled.

If the variables and coefficient are integers, we can formulate a complement. For the constraint $x_1 + 2x_2 \leq 10$, the violation is $x_1 + 2x_2 \geq 11$. (We cannot use strict inequalities for linear programming).



Problem

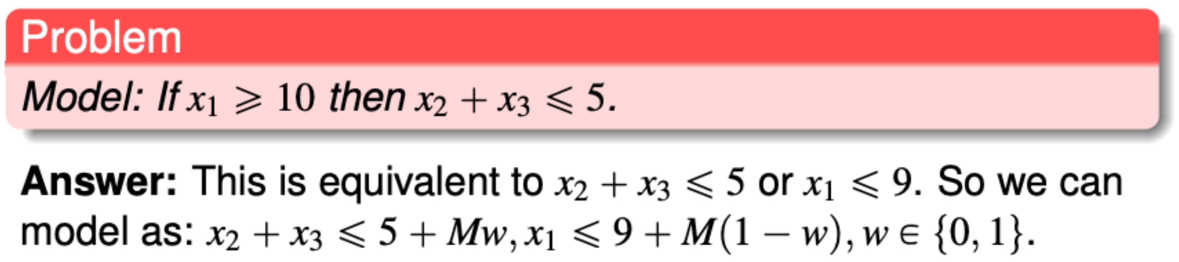
Model: either $x_1 + 2x_2 \leqslant 10$ *or* $x_1 - x_3 \geqslant 5$*, but not both.*

Answer: This is equivalent to: $x_1 + 2x_2 \leqslant 10$ and $x_1 - x_3 \leqslant 4$, or $x_1 + 2x_2 \geqslant 11$ and $x_1 - x_3 \geqslant 5$. So we can model as:

$$
\begin{aligned}
x_1 + 2x_2 \leqslant 10 + Mw, x_1 - x_3 \leqslant 4 + Mw \\
x_1 + 2x_2 \geqslant 11 - M(1-w), x_1 - x_3 \geqslant 5 - M(1-w) \\
w \in \{0,1\}
\end{aligned}
$$

Implication statement into logical constraint

If $P \implies Q$, $\overline{P} \vee Q$ must be true. $\overline{P}$ is the complement of P.



Problem

Model: If $x_1 \geqslant 10$ *then* $x_2 + x_3 \leqslant 5$.

Answer: This is equivalent to $x_2 + x_3 \leqslant 5$ or $x_1 \leqslant 9$. So we can model as: $x_2 + x_3 \leqslant 5 + Mw, x_1 \leqslant 9 + M(1-w), w \in \{0,1\}$.

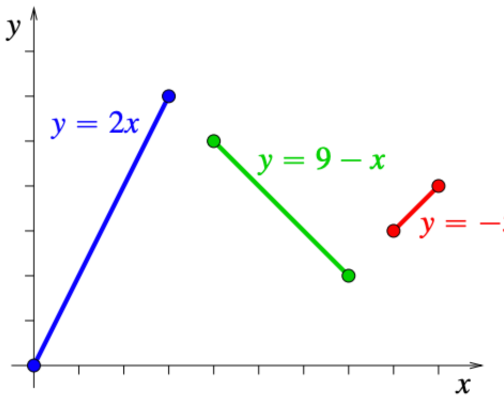
Piecewise linear functions

This is more for objective function, though it is possible for the constraint.

Piecewise linear functions

For some problems, linear functions are not appropriate in the model. Piecewise linear functions significantly increase modeling power.

Assume x is integer. We want to model this function:



The final model

$$w_1 = \begin{cases} 1 & \text{if } 0 \leqslant x \leqslant 3 \\ 0 & \text{otherwise} \end{cases} \quad w_2 = \begin{cases} 1 & \text{if } 4 \leqslant x \leqslant 7 \\ 0 & \text{otherwise} \end{cases} \quad w_3 = \begin{cases} 1 & \text{if } 8 \leqslant x \leqslant 9 \\ 0 & \text{otherwise} \end{cases}$$

$$x_1 = \begin{cases} x & \text{if } 0 \leqslant x \leqslant 3 \\ 0 & \text{otherwise} \end{cases} \quad x_2 = \begin{cases} x & \text{if } 4 \leqslant x \leqslant 7 \\ 0 & \text{otherwise} \end{cases} \quad x_3 = \begin{cases} x & \text{if } 8 \leqslant x \leqslant 9 \\ 0 & \text{otherwise} \end{cases}$$

This can be modeled with the following constraints:

$$0w_1 \leqslant x_1 \leqslant 3w_1, \qquad 4w_2 \leqslant x_2 \leqslant 7w_2, \qquad 8w_3 \leqslant x_3 \leqslant 9w_3,$$

$$x = x_1 + x_2 + x_3, \qquad w_1 + w_2 + w_3 = 1,$$

$$y = (0w_1 + 2x_1) + (9w_2 - x_2) + (-5w_3 + x_3)$$

$$w_1, w_2, w_3 \in \{0, 1\}, \qquad x_1, x_2, x_3 \text{ integer}$$

x is replicated into three parts. x_i will be set to zero if we are not using the i-th piece. w_i indicates whether are we using the i-th piece.

LP relaxations to solve IP

In this approach, we solve the IP by solving its LP relaxations. The relaxation is tighted with new constraints until we obtain the solution for the IP.

LP relaxation

You solve the LP relaxation. If the optimum solution is not all integers, you find valid inequalities to make the corner points integer.

- A **valid inequality** for an integer program is any inequality that is satisfied by all of its feasible solutions.

Knapsack cover inequality

You observe a subset S with $|S|$ elements such that its weights is larger than the limit. The **knapsack cover inequality** prevents you from choosing all of these elements, you can choose at most $|S|-1$ elements from S.

$$\sum_{j\in S} x_j \le |S|-1$$

A **minimal cover** is a cover that $|S|-1$ of its elements has a weight below the limit. Using minimal covers is usually useful.

Gomory cuts

Suppose all variables are integers. You have a equality constraint with deciment coefficient and RHS.

- You can round down the coefficients and you obtain a inequality.
- You can round down the RHS because the LHS can only provide integers.
- Take the difference of the equality from the RHS and you obtain a Gomory cut.
    - "It is a cutting plane that cuts off the BFS associated with the tableau"
    - The new inequality is added into the LP, and process repeats until the IP is solved (???)

Branch-and-Bound

(incomplete and may be inaccurate)

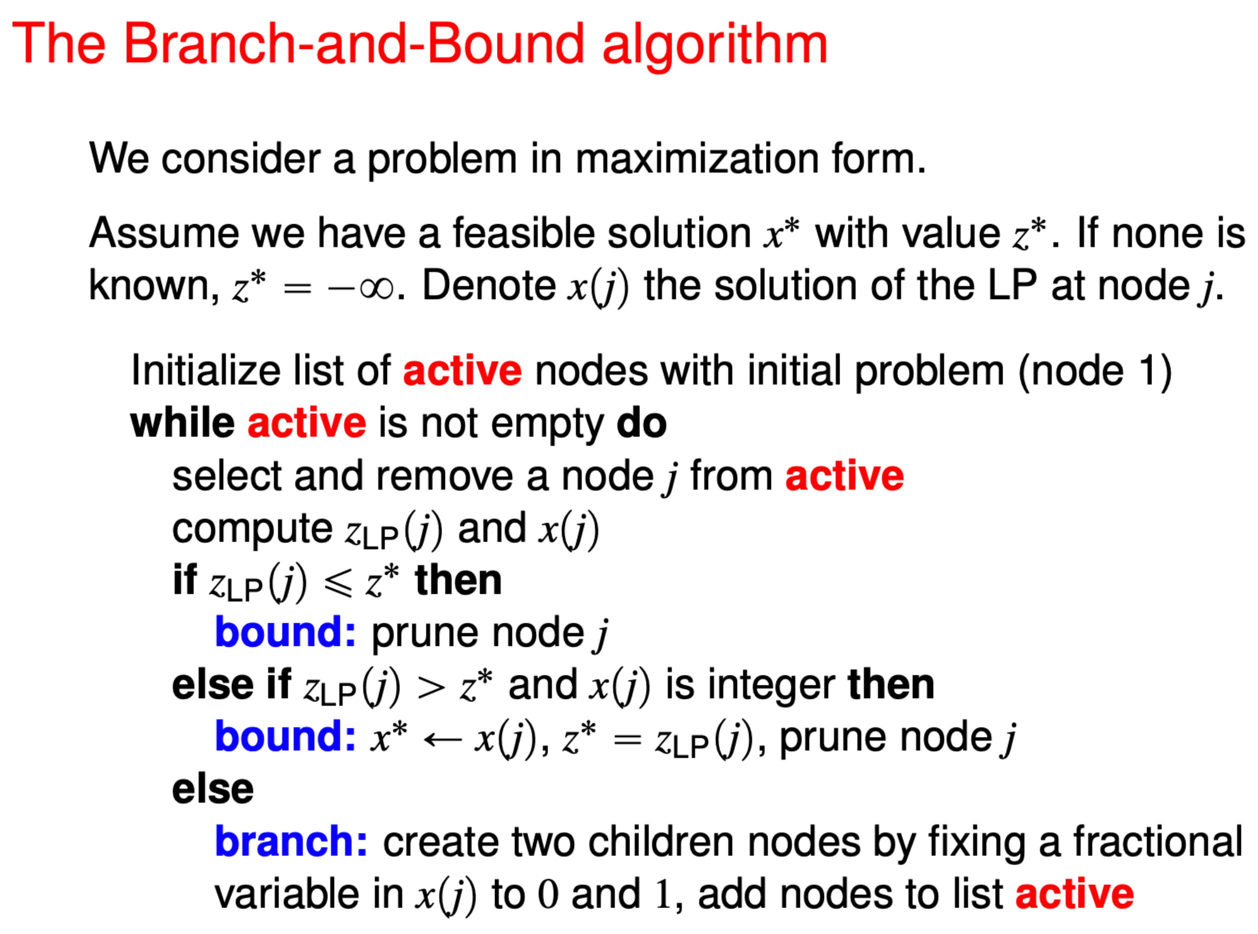
For a (**minimising**) **binary/integer program** with n variables, the brute-force or full enumeration takes 2^n computations. We do not want to do all of them.

Key idea - prune the tree

- Any integer solution produced is considered for the lower bound of the IP. If LP optimum of any subproblem is lower than the lower bound prune.
- If the LP solution of a subproblem is integer, do you not need to continue branching.

Properties

- The IP optimum is the minimum of the children IP problems (i.e. equal to one of them)
- The LP optimum is smaller than the children LP optimums (may not be equal to one of them).



The Branch-and-Bound algorithm

We consider a problem in maximization form.

Assume we have a feasible solution x^* with value z^*. If none is known, $z^* = -\infty$. Denote $x(j)$ the solution of the LP at node j.

```
Initialize list of active nodes with initial problem (node 1)
while active is not empty do
    select and remove a node $j$ from active
    compute $z_{LP}(j)$ and $x(j)$
    if $z_{LP}(j) \leqslant z^*$ then
        bound: prune node $j$
    else if $z_{LP}(j) > z^*$ and $x(j)$ is integer then
        bound: $x^* \leftarrow x(j)$, $z^* = z_{LP}(j)$, prune node $j$
    else
        branch: create two children nodes by fixing a fractional
        variable in $x(j)$ to $0$ and $1$, add nodes to list active
```

Dynamic programming

Elements of a Dynamic Program

- **Stages.** the decision to be optimized are taken sequentially over a number of stages, that typically represent time periods.
- **States.** at each stage, all the information required to take future decisions can be represented by a state, regardless of how we reached the current state.
- **Decisions.** at each stage and state, we have a set of decisions or actions available, which bring us to some state in the subsequent stage.
- **Principle of optimality.** any optimal sequence of decisions has the property that, whatever the current stage, state and decision, the remaining decisions must be optimal with regard to the stage and state resulting from the current decision. (Is this the optimal substructure?)

Essentially, you need to be able to calculate for all the states in one direction until you get your required answer.

Construction

You need to define the **stages** and the **states**. This breaks down the problem into subproblems.

Each stage k consists of all the states. Each stage may only have one state (126 match game) or many states (shortest path - the current node). At each stage, each state has a **value** $f_k(j)$ (126 match game - win or lose, shortest path - cost to reach)

You use the past stage(s) to compute the next stage. This is the **recursion**.

Your program needs to stop when you have obtained the answer, you need to specify a **stopping condition**. The problem may be **unbounded** and your program need to be able to detect and return that.

Preflight checklist

Please provide the **mathematical expression** to compute the next stage.

Please remember to define the **boundary conditions** of all states.

126 Match game

Each **stage** k is the number of games played.

There is only one **state** for each stage. So we not specify the state.

The **value** function $f(k)$ indicates whether player one will win if the game starts with n matches.

The **recursion** $f(k) = 1 - \min\{1, f_{k-1}, f_{k-2}, f_{k-6}\}$

The **boundary condition** $f(1) = f(2) = f(4) = f(5) = f(6) = 1$ and $f(3) = 0$.

The **stopping condition** when $f(n)$ is computed

Shortest path

Each **stage** k is maximum number of arcs allowed to each the node.

Each **state** j is the current node.

The **value** function $f_k(j)$ is the cost of the path from the starting node to the node ended.

The **recursion** $f_k(j) = \min\{f_{k-1}(j), \min_{(i,j)\in A}\{f_{k-1}(i) + c_i j\}\}$ You either do not move, or you move from another location and incur the cost.

The **boundary condition** $f_0(1) = 0$ and for **other node**s $f_0(j) = \infty$

The **stopping condition** $f_k(j) = f_{k+1}(j)\ \forall j$. If $k > 2J$ there is negative cycle.

Optimal capacity expansion

Each **stage** k is the year.

Each **state** j is the number of plants at the beginning of the year.

The **value** function $f_k(j)$ is minimum cost to adhere to the requirement from stage k until the end of the planning horizon, if state j plants has already been built.

The **recursion** $f_k(i) = \min_{j=1,2,3}\{f_{k+1}(i), 1.5 + c_k j + f_{k+1}(i+j)\}$ but ∞ if $i < d_k$. You either do not build a plant, or you build a number of plants incurring fixed and variable cost.

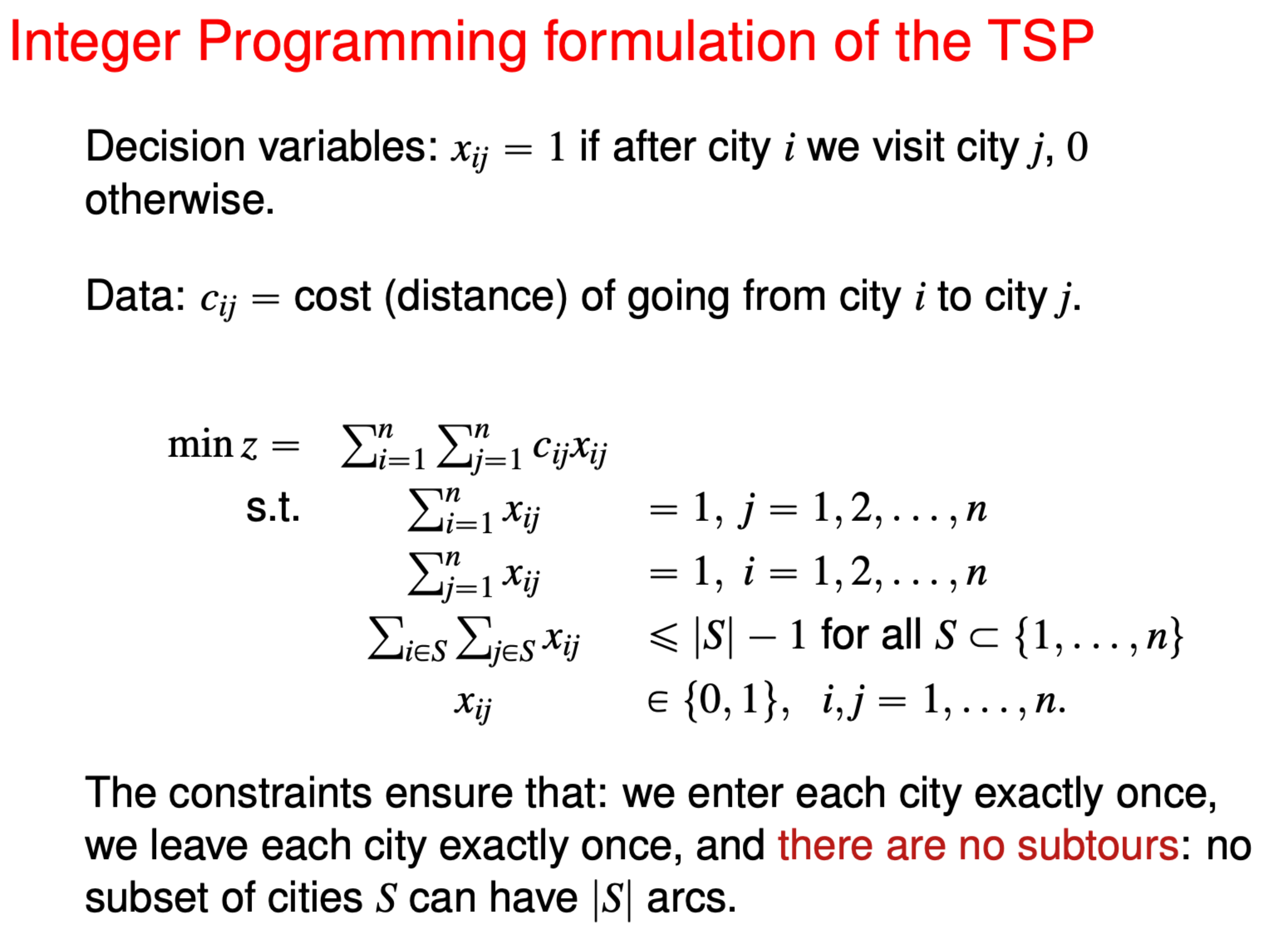
The **boundary condition** $f_6(8) = 0$, $f_6(j) = \infty$ for $j \leq 7$.

The **stopping condition** if $f_0(0)$ has been computed. (This is an example of backward recursion.)

Travelling salesman problem

Find a tour in the graph that has the minimum cost.

Integer programming formulation



Integer Programming formulation of the TSP

Decision variables: $x_{ij} = 1$ if after city i we visit city j, 0 otherwise.

Data: $c_{ij} =$ cost (distance) of going from city i to city j.

$$
\begin{array}{rll}
\min z = & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} & \\
\text{s.t.} & \sum_{i=1}^{n} x_{ij} & = 1,\ j = 1, 2, \ldots, n \\
& \sum_{j=1}^{n} x_{ij} & = 1,\ i = 1, 2, \ldots, n \\
& \sum_{i \in S} \sum_{j \in S} x_{ij} & \leqslant |S| - 1 \text{ for all } S \subset \{1, \ldots, n\} \\
& x_{ij} & \in \{0, 1\},\quad i, j = 1, \ldots, n.
\end{array}
$$

The constraints ensure that: we enter each city exactly once, we leave each city exactly once, and there are no subtours: no subset of cities S can have $|S|$ arcs.

To eliminate all subtours you need 2^n constaints. At some n you cannot even insert the constraints into your computer.

Subtour elimination

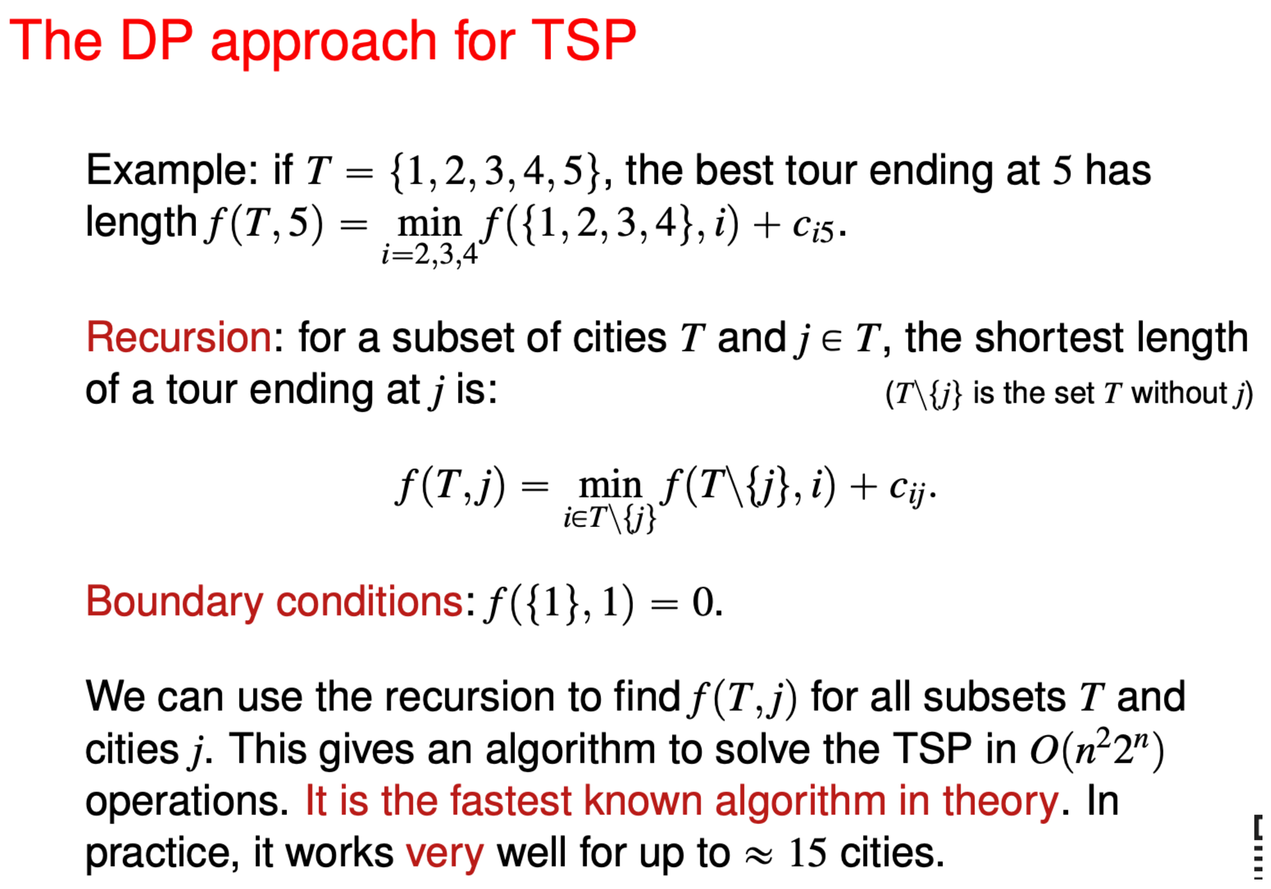
You solve without the subtour constraint initially. When the solution contains a subtour, add the subtour into the constraint and solve again. You might need to do this many times. Nevertheless each solution provides a lower bound to the solution. (Any feasible solution provides an upper bound to the problem.)

Dynamic programming

Assume that we know the best tour from one node to another, excluding certain nodes. This can be the basis for iteration.

We say that subtour T **dominates** T' if:

1. T and T' begin at 1 and end at the same city j.
2. The length of T is at most the length of T' .
3. T and T' visit the same set of cities.



The DP approach for TSP

Example: if $T = \{1,2,3,4,5\}$, the best tour ending at 5 has length $f(T,5) = \min_{i=2,3,4} f(\{1,2,3,4\},i) + c_{i5}$.

Recursion: for a subset of cities T and $j \in T$, the shortest length of a tour ending at j is: ($T\backslash\{j\}$ is the set T without j)

$$f(T,j) = \min_{i \in T\backslash\{j\}} f(T\backslash\{j\},i) + c_{ij}.$$

Boundary conditions: $f(\{1\},1) = 0$.

We can use the recursion to find $f(T,j)$ for all subsets T and cities j. This gives an algorithm to solve the TSP in $O(n^2 2^n)$ operations. It is the fastest known algorithm in theory. In practice, it works very well for up to ≈ 15 cities.

Heuristic approaches

A heuristic is an algorithm that will find some solution that may not be optimal.

Provable guarantees

- "k-approximation algorithm" is at most k times worse than the optimum.
- "domination number D" is at least as good as D of the possible solutions of of $(n-1)!$ solutions.

Method	**Runtime**	k	D
Nearest Neighbour	$O(n^2)$	NA	1
Greedy	$O(n^2 log(n))$	NA	1
Insertion	$O(n^2)$	NA	$(n-2)!$
Christofiedes	?	2*	1
Local search (2-Opt)	$O(n^2)$	NA	$(n-2)!$

Nearest Neighbour

Start from any node - this is random. (You can try all of them.) Then, always go to the node with smallest distance to the current node and that has not been previously visited. After visiting all nodes, go back to the starting node.

Greedy Algorithm

You start with the cheapest arc, and go to the next cheapest node that do not create a small cycle, until all nodes is visited.

Insertion Algorithm

Randomly start with two nodes, you have a 2-node subtour. You randomly choose a node to insert. You make a 3-node subtour by finding the best edge to replace. Repeat until you cover the entire map.

Christofiedes Algorithm

Start with the minimum spanning tree. Then do magic.

Assumes **triangle inequality**. (Triangle inequality do not hold in real life, however. A direct path may be shorter than a detour.)

Local search Algorithms (2-Opt)

Find a pair of edges that will have a lower cost when swapped. Repeat until you cannot find such a pair.

This improves any current solution. This algorithm makes it harder for observers to point out an easier solution visually.