Further Algorithms on Graph Problems

a. Network flows
b. Local search and meta-heuristics

Lau Hoong Chuin
School of Computing

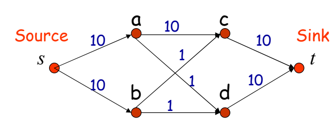
Network Flows

- One major class of real-world problems involves moving something through a network, from a source to a destination, subject to capacity restrictions in the various network connections
  - Moving oil from a field to a refinery
  - Moving products from a factory to a warehouse
  - Moving data through the Internet

Network Flows Applications

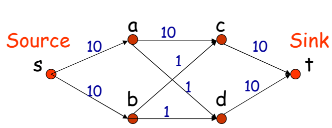
Network	Nodes	Arcs	Flow
communication	telephone exchanges, computers, satellites	cables, fiber optics, microwave relays	voice, video, packets
circuits	gates, registers, processors	wires	current
mechanical	joints	rods, beams, springs	heat, energy
hydraulic	reservoirs, pumping stations, lakes	pipelines	fluid, oil
financial	stocks, currency	transactions	money
transportation	airports, rail yards, street intersections	highways, railbeds, airway routes	freight, vehicles, passengers
chemical	sites	bonds	energy

Network Flows

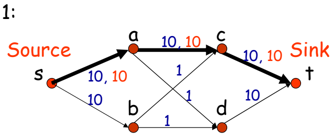


- A network is a weighted directed graph where
  - There is exactly one vertex with an indegree of 0 (source)
  - There is exactly one vertex with an outdegree of 0 (sink)
  - each edge (v,w) has a capacity $cap(v,w)$
- ***Capacity constraint:*** A *flow* is a function *f*, that is chosen for each edge so that $f(v,w) \le cap(v,w)$.
- ***Flow conservation:*** For all *v* except s and t, $\sum_{v \in V} f(u,v) = 0$
- Objective: to maximize the flow allocation

Example
Finding max flow by inspection



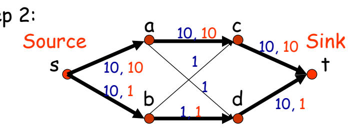
Step 1:



Flow is of size 10

Example
Finding max flow by inspection

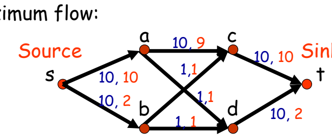
Step 2:



Flow is of size 10+1 = 11

Not obvious

Maximum flow:



Flow is of size 10+2 = 12

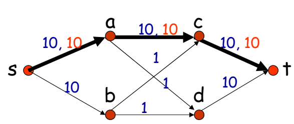
Ford-Fulkerson Algorithm
Augmenting paths

1. initialize flow f to 0
2. **while** there exists an augmenting path p
3. **do** augment flow f along p
4. **return** f

Ford-Fulkerson Algorithm
Augmenting paths

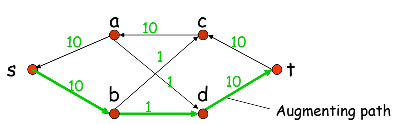
1. Set $f(v,w) = -f(w,v)$ on all edges.
2. Define a residual network in which $res(v,w) = cap(v,w) - f(v,w)$
3. Find paths from s to t for which there is positive residue.
4. Increase the flow along the paths to augment them by the minimum residue along the path.
5. Keep augmenting paths until there are no more to augment.

Example of Residual Network



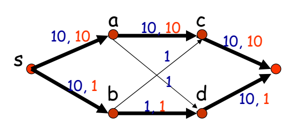
Flow is of size 10

residual network $res(v,w) = cap(v,w) - f(v,w)$



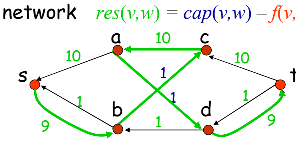
Example of Residual Network

Step 2:



Flow is of size 10+1 = 11

residual network $res(v,w) = cap(v,w) - f(v,w)$



Flows and Cuts

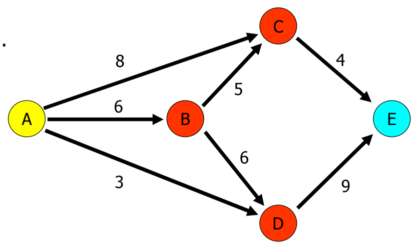
- Proof of optimality of augmenting path algorithm is based on the notion of cuts
- A cut in a network is a partition of its vertices into two sets, S and T, such that
  - The source vertex lies in S
  - The sink vertex lies in T
- The sum of capacity of all arcs crossing a vertex in S with a vertex in T is the capacity of the cut, cap(S,T)
- For any given cut, the flow in a transportation network equals the total flow along arcs from S to T, minus the total flow along arcs from T to S

Flows and Cuts

- The maximum flow of a network cannot exceed the capacity of *any* cut within that network
- The maximum flow of a network equals the minimum capacity of all cuts
- Let f be a flow, and let (S, T) be a cut. If |f| = cap(S,T), then f is a max flow and (S, T) is a min cut.

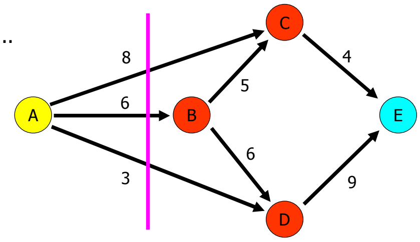
Max-Flow Min-Cut

Available cuts...
A / BCDE
AC / BDE
AB / CDE
AD / BCE
ABC / DE
ABD / CE
ABCD / E



Max-Flow Min-Cut

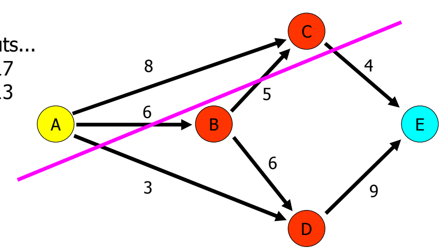
Available cuts...
A / BCDE: 17
AC / BDE
AB / CDE
AD / BCE
ABC / DE
ABD / CE
ABCD / E



8+6+3=17

Max-Flow Min-Cut

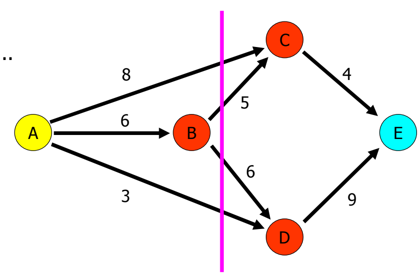
Available cuts...
A / BCDE: 17
AC / BDE: 13
AB / CDE
AD / BCE
ABC / DE
ABD / CE
ABCD / E



4+6+3=13

Max-Flow Min-Cut

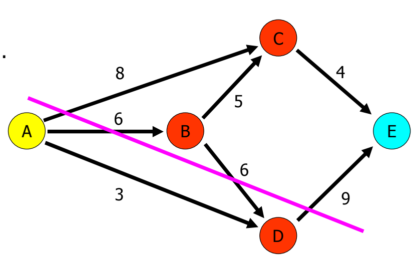
Available cuts...
A / BCDE: 17
AC / BDE: 13
AB / CDE: 22
AD / BCE
ABC / DE
ABD / CE
ABCD / E



8+5+6+3=22

Max-Flow Min-Cut

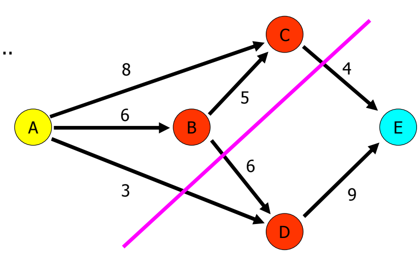
Available cuts...
A / BCDE: 17
AC / BDE: 13
AB / CDE: 22
AD / BCE: 23
ABC / DE
ABD / CE
ABCD / E



8+6+9=23

Max-Flow Min-Cut

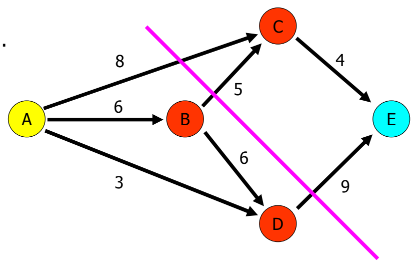
Available cuts...
A / BCDE: 17
AC / BDE: 13
AB / CDE: 22
AD / BCE: 23
ABC / DE: 13
ABD / CE
ABCD / E



4+6+3=13

Max-Flow Min-Cut

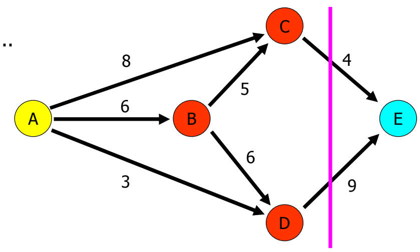
Available cuts...
A / BCDE: 17
AC / BDE: 13
AB / CDE: 22
AD / BCE: 23
ABC / DE: 13
ABD / CE: 22
ABCD / E:



8+5+9=22

Max-Flow Min-Cut

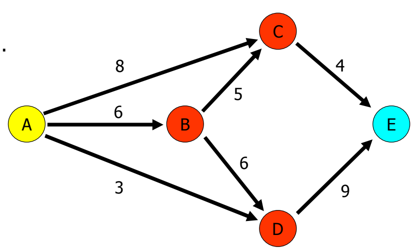
Available cuts...
A / BCDE: 17
AC / BDE: 13
AB / CDE: 22
AD / BCE: 23
ABC / DE: 13
ABD / CE: 22
ABCD / E: 13



4+9=13

Max-Flow Min-Cut

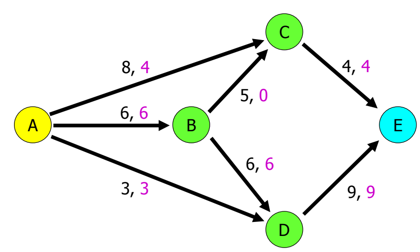
Available cuts...
A / BCDE: 17
AC / BDE: 13
AB / CDE: 22
AD / BCE: 23
ABC / DE: 13
ABD / CE: 22
ABCD / E: 13



Examining all cuts, min cut is 13, max flow is 13

Max-Flow Min-Cut

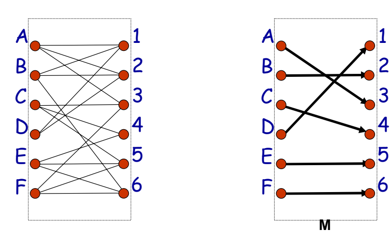
Solution



History of Max Flow Algorithms

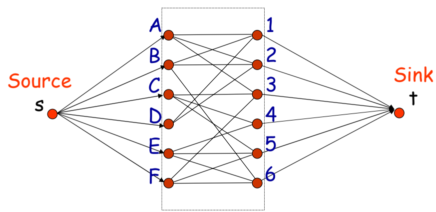
Year	Discoverer	Method	Big-Oh
1951	Dantzig	Simplex	mn^2U
1955	Ford, Fulkerson	Augmenting path	mnU
1970	Edmonds-Karp	Shortest path	m^2n
1970	Dinitz	Shortest path	mn^2
1972	Edmonds-Karp, Dinitz	Capacity scaling	$m^2 \log U$
1973	Dinitz-Gabow	Capacity scaling	$mn \log U$
1974	Karzanov	Preflow-push	n^3
1983	Sleator-Tarjan	Dynamic trees	$mn \log n$
1986	Goldberg-Tarjan	FIFO preflow-push	$mn \log (n^2 / m)$
. . .	. . .	. . .	. . .
1997	Goldberg-Rao	Length function	$m^{3/2} \log (n^2 / m) \log U$ $mn^{2/3} \log (n^2 / m) \log U$

Bipartite Matching as Max Flows



- Given an undirected graph $G=(V, E)$, a matching M is a subset of edges such that for all vertices, at most one edge of M is incident on v.
- A bipartite graph is a graph such that the set of vertices can be partitioned into two sets A, B and every edge has a vertex in A and another in B.

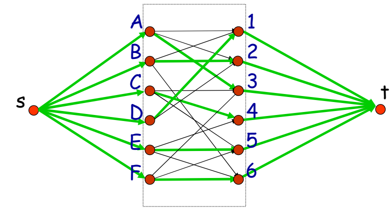
Bipartite Matching as Max Flows



- How to find the maximum bipartite matching?

  equivalent to solving a network flow problem with capacities of 1!

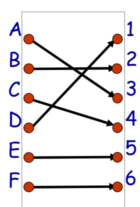
Bipartite Matching as Max Flows



- If two edges (i,j) and (k,j) share a node, then either x_{ij} = 0, or x_{kj} = 0, or both. Otherwise the arc capacity of (j,t) will be violated.
- If two edges (i,j) and (i,k) share a node, then either x_{ij} = 0, or x_{ik} = 0, or both.

Bipartite Matching as Max Flows

Maximum Size Matching:



Exercise (Parallel Machines Scheduling)

Given *M* parallel machines, a set of *n* jobs, each having a length, release time and deadline. A machine can work on only one job at a time. A job can be served by only one machine at a time.

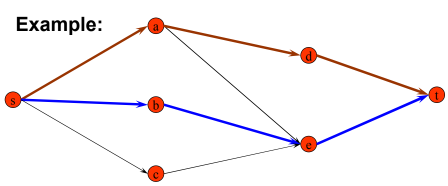
Preemption allowed, i.e., service of a job can be interrupted, and jobs can switch machines.

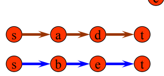
Is there a schedule for completing every job within its deadline?

Exercise (The Messenger Problem)

Problem: Given a network, find the largest number of messengers that can be sent through the network where the paths are disjoint (except at the ends).

Example:





Here is an optimal solution with 2 messengers.

Local Search and Meta-Heuristics

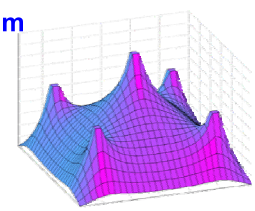
Objectives:

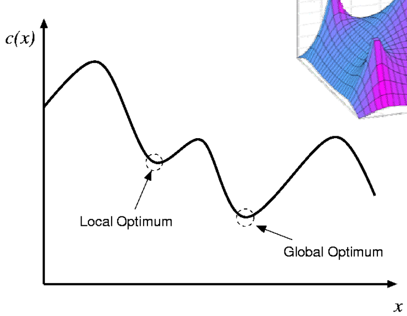
- To learn the basic Local Search paradigm
- To apply LS to solve (NP-hard) optimization problems
- To gain preliminary understanding of sophisticated LS-based meta-heuristics such as tabu search and genetic algorithms

Heuristics

- Optimization problem:
  $\min_{x \in S} f(x)$, where S=feasible solutions space
  (find a feasible x such that f is minimized)
- Heuristic algorithms do not guarantee to find an optimal solution (as opposed to exact algorithms)
- However, they are usually designed to find a "reasonably good" solution quickly.
- Solutions to optimization problems:
  – Global Optimum: Equal/better than all other solutions
  – Local Optimum: Equal/better than all solutions in a certain neighborhood

Global vs. Local Optimum





Why Do We Need Heuristics?

- Many real world applications are too large to be solved by exact methods (e.g. Branch and Bound, Cutting Planes, etc.)
- Solutions are usually needed fast (sometimes even in real time)
  – for **hard** problems, the time for finding an optimal solution is usually much greater than finding a near-optimal solution
- Optimal solutions, although desirable, are often not needed in the real world.

Local Search

Definitions:

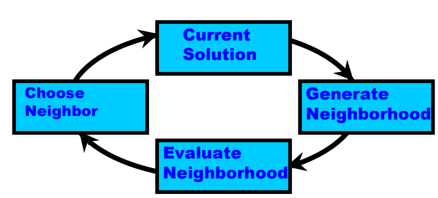
- N (neighborhood) : $S \rightarrow 2^S$
- The size of a neighborhood is the number of solutions in the neighborhood set

Algorithm:

1. Pick a starting solution s // by other algorithm
2. If $f(s) < \min_{s' \text{ in } N(s)} f(s')$, stop // what is the time complexity?
3. Else set s = s' s.t. $f(s') = \min_{s'' \text{ in } N(s)} f(s'')$
4. Goto Step 2

Local Search

An iteration in a local search

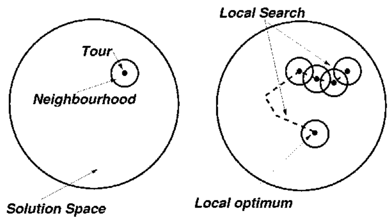


Example: Traveling Salesman Problem

Given:

- A set of n cities $\{c_1, c_2, \ldots, c_n\}$
- Cost matrix, containing cost to travel between cities

Find a minimum-cost tour.



Solving TSP Heuristically

3 Broad Steps:

1. Construction Heuristics
   – Constructing a tour from scratch
2. **Local Search Algorithms**
   – Improving an existing tour
3. Extensions (not covered)
   – "Escaping" from locally optimal tours

Construction Heuristics

- Build a tour one city at a time in a Greedy fashion
- Popular methods:
  - Nearest Neighbour heuristic
  - Multiple Fragment heuristic
  - Insertion heuristic
  - Savings heuristic
  - Spanning tree heuristic, etc.

Nearest Neighbour

Step 1: Choose starting city p at random

Step 2: Scan remaining cities for the city NN_p nearest to p

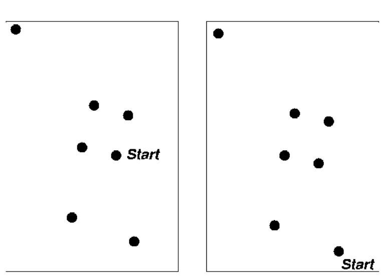
Step 3: Add edge (p, NN_p) to the sequence

Step 4: Set $p = NN_p$

Step 5: Repeat from Step 2 until all cities are added

Step 6: Add last edge to complete a tour

Nearest Neighbour Tours



Nearest Neighbour Tours

Some Variants of Nearest Neighbour

- Double-Sided Nearest Neighbour
  – Grow the nearest neighbour sequence from both ends
- Randomized Nearest Neighbour
  – Instead of using the nearest neighbour of a city, pick a neighbour at random from the set of remaining k nearest neighbours of the city and add it to the sequence

Multiple Fragment Heuristic

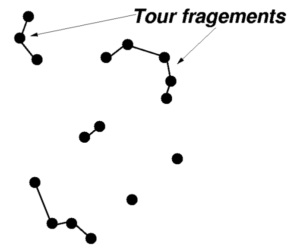
Basic Idea:

Build a tour one edge at a time, by adding the shortest remaining available edge to the tour edge set.

Step 1: Order the set of edges in increasing length in a list
Step 2: Pick first *available* edge on the list
Step 3: Remove edge from list and add it to the tour edge set
Step 4: Repeat from step 2 until there are no more *available* edges on the list

Note: similarity with Kruskal's algorithm for MSTs

Multiple Fragment Tour:



Multiple Fragment Heuristic

Insertion Heuristics

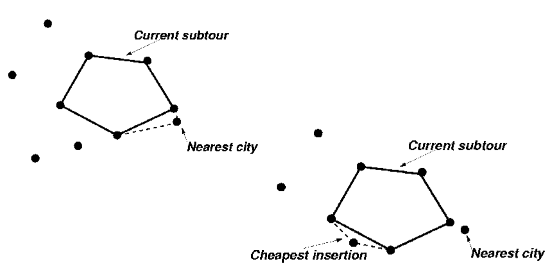
- Basic Idea:
  Starting with a subtour made of 2 arbitrary chosen cities grow a tour by inserting cities which fulfill some criteria.
- The criteria usually depends on the cities which are already in the tour

Step 1: Form a subtour by choosing 2 cities at random
Step 2: Scan remaining cities for the city which fulfills insertion criteria
Step 3: Insert the city to the subtour
Step 4: Repeat Step 2 until all cities are in the tour

Some Insertion Criteria

- Nearest Insertion:
  - Insert the city that has shortest distance to a tour city
- Farthest Insertion:
  - Insert the city whose shortest distance to a tour city is maximized
- Cheapest Insertion:
  - Choose the city whose insertion causes the least increase in length of the resulting subtour
- Random Insertion
  - Select the city to be inserted at random

Nearest vs Cheapest Insertion



Analysis

Number of steps:

- *n-2* cities are to be inserted
- For each insertion the set of remaining cities has to be scanned : *O(n)*

Worst case run time: $O(n^2)$

Applications of Local Search

- Local Search Algorithms for TSP
- Local Search Algorithm for Bin Packing
- Advanced Local Search variants
  - Tabu search
  - Simulated annealing
  - Genetic algorithms

Local Search Algorithm for TSP

- Create an initial tour // discussed previous lecture
- Define a local search neighborhood
- Two tours are neighbors if they share most of their edges

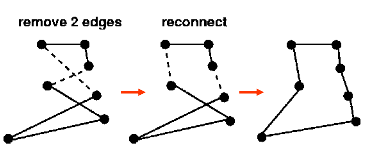
- Edge exchange neighborhood:

> Delete *k* edges in the current tour and then add *k* edges which form a new feasible tour

This neighborhood is called *k*-opt.

2-opt

- Delete 2 edges, add 2 edges to restore the tour



- 2-opt removes the “crossings” of edges in a tour

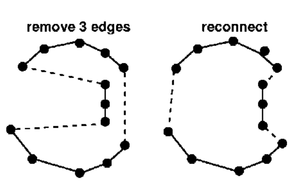
2-opt Analysis

- How many possible ways?
- Size of the 2-opt neighborhood:
  - Number of edge pairs = $n(n-3)/2$
  - Exactly 1 way to restore tour
- At each step of Local Search, evaluate $O(n^2)$ new tours and choose the best available.

  ⇒ Worst-case run time: $O(n^2)$

3-opt

- Delete 3 edges, add 3 edges to restore the tour

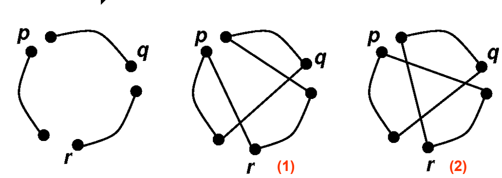


- 3-opt can remove subsequences of the tour

3-opt Analysis

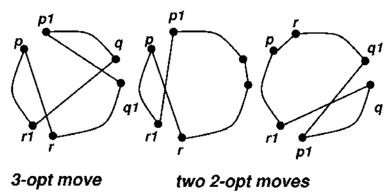
- Size of the 3-opt neighborhood:
  - Number of edge triples = $\binom{n}{3}$
  - Each of which has 8 possible ways to be restored
  - Some neighbors may not be valid
- At each step of Local Search, $O(n^3)$ new tours

  ⇒ worst case run time: $O(n^3)$



k-opt

- It may seem sensible to increase k even further.
- However, once k edges of a tour have been deleted there are $O(n^k)$ new tours. Expensive!
- Observe that some 3-opt moves are equivalent to applying two 2-opt moves

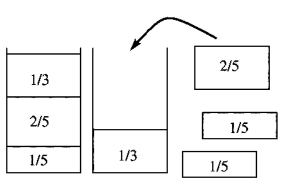


Lin-Kernighan Heuristic

- First observed by Lin and Kernighan
- One of most successful heuristics to solve combinatorial optimization problems
- "Adaptive" *k*-Opt
- Build *k*-opt moves (with variable *k*) by a sequence of 2-opt moves
- Take the sequence of the first *k* swaps that **most** improves the solution

Bin Packing Problem

- One-dimensional bin packing problem
- Given:
  - n items $a_1, a_2, \ldots, a_n$, $0<size(a_i)\leq 1\ \forall\ i=1,..,n$
  - unlimited number of bins of size 1
- Goal: Pack all items into a minimum number of bins



Bin Packing Algorithms

- Although the bin packing problem is an NP-hard problem, 2 very successful heuristics have been proposed:

1. First Fit Decreasing Heuristic
2. Best Fit Decreasing Heuristic

The First Fit Decreasing (FFD) heuristic

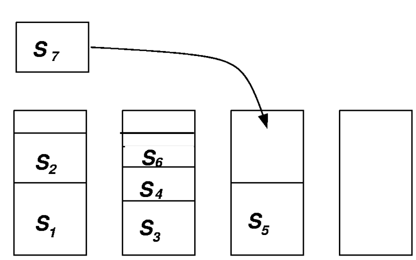
Step 1: Arrange items in decreasing size

Step 2: Assign the first item on the list to the first bin it can fit into

Step 3: Remove item from the list and reduce capacity of the selected bin

Step 4: Repeat from step 2 until all items have been assigned

First Fit Strategy:



First Fit Strategy

What's wrong with this diagram?

The Best Fit Decreasing (BFD) heuristic

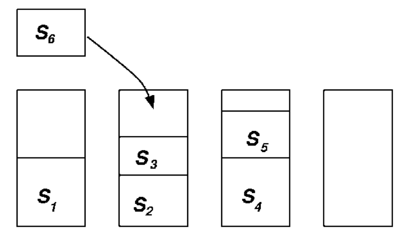
Step 1: Arrange items in decreasing size

Step 2: Assign the first item on the list to the bin it fits best, i.e. fills up to the highest level.

Step 3: Remove item from the list and reduce capacity of the selected bin

Step 4: Repeat from Step 2 until all items have been assigned

Best Fit Strategy:



Best Fit Strategy

Local Search Algorithm for Bin Packing

- Objective: decrease the number of bins needed

- "Improvement" strategy:
  modify packing so that the packing height in the emptiest bin is reduced

Neighborhoods

For bins *i* and *j*, define:

- 1-1-exchange neighborhood:
  swap one item in bin *i* with an item in bin *j*
- p-q-exchange neighborhood:
  Swap *p* items in bin *i* with *q* items in bin *j*

Note: Not all exchanges lead to a feasible packing

Local Search Algorithm

- **Step 1:** Create an initial packing
- **Step 2:** Select two bins (at random or following some rule)
- **Step 3:** Scan chosen bins for a *p*-*q*-exchange that "improves" the packing
- **Step 4:** Repeat from Step 2 until no improvement is found

Advanced Local Search Paradigms (Meta-heuristics)

- Tabu Search
- Simulated Annealing
- Evolutionary (Genetic) Algorithms
- Ant Colony Optimization
- etc.

Tabu Search

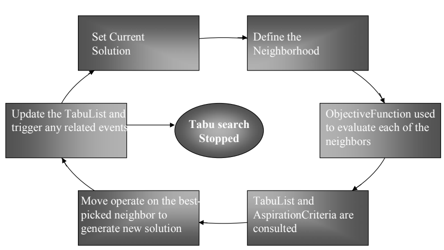
- A tabu list is maintained for forbidden (tabu) moves, to avoid cycling
- Tabu moves are based on the short- and long-term history of the search process.
- Aspiration criteria is the condition which allows the tabu status of a tabu move to be overwritten so that the move can be considered at the iteration.
- The next move is the best move among the feasible moves from the neighborhood of the current solution. A tabu move is taken if it satisfies the aspiration criteria.

Tabu Search Algorithm

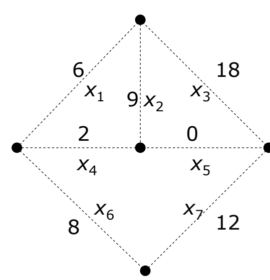
- s: *current solution*
- $N(s)$: *neighborhood set* of s where $N(s) \subseteq S$
- $T(s)$: *tabu set* of s where $T(s) \subseteq N(s)$
- $A(s)$: *aspiration set* of s where $A(s) \subseteq T(s)$

1. construct an initial solution s
2. while *not finished*
3. compute $N(s)$, $T(s)$, $A(s)$
4. choose $s' \in (N(s) - T(s)) \cup A(s)$ s.t. cost(s') is min
5. $s = s'$
6. endwhile

Tabu Search Algorithm



Example: Constrained MST Problem



Constraints:

(1) $x_1 + x_2 + x_6 \leq 1$

(2) $x_1 \leq x_3$

Penalty of each constraint violation= 50

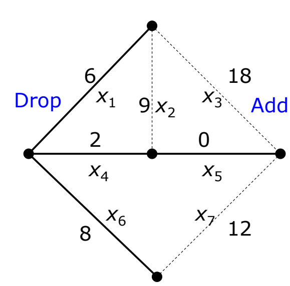
Example

- Neighborhood: standard "edge swap"
- An edge is tabu if it was added within the last two iterations
- The aspiration criteria is satisfied if the tabu move would create a tree that is better than the best tree so far

Example

- Iteration 1 (initial MST)

$cost = 16 + 100$



Constraints:

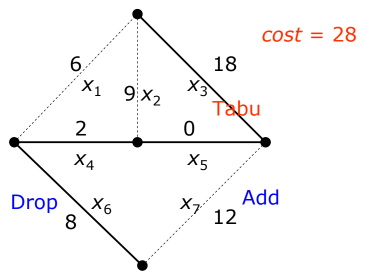
(1) $x_1 + x_2 + x_6 \leq 1$

(2) $x_1 \leq x_3$

Penalty of each constraint violation= 50

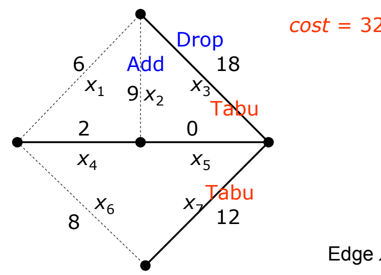
Example

- Iteration 2



Example

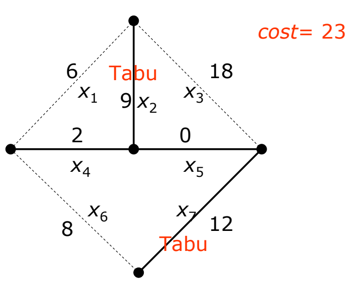
- Iteration 3



Edge x_3 is aspired.

Example

- Iteration 4



Genetic Algorithms

Components:

- Encoding / solution *(gene, chromosome)*
- Initialization procedure *(creation)*
- Evaluation function *(fitness in environment)*
- Selection of parents *(reproduction)*
- Genetic operators *(mutation, crossover)*

Standard Genetic Algorithm

```
initialize population;
evaluate population;
while TerminationCriteriaNotSatisfied
{
   select parents for reproduction;
   perform genetic operations to generate
     offspring (new population)
   evaluate population;
}
```


TSP Example

Encoding is an ordered list of city numbers

1) London	3) Dunedin	5) Beijing	7) Tokyo
2) Venice	4) Singapore	6) Phoenix	8) Victoria

CityList1 (3 5 7 2 1 6 4 8)

CityList2 (2 5 7 6 8 1 3 4)

Crossover

Parent1	(3 5 7 2 1 6 4 8)
Parent2	(2 5 7 6 8 1 3 4)
Child	(2 5 7 2 1 6 3 4)

Note: may generate infeasible offspring!

Crossover

Cycle crossover:

Parent1	(2 3 5 6 4 1 7 8)
Parent2	(1 4 2 3 6 5 8 7)
Child1	(1 3 2 6 4 5 8 7)
Child2	(2 4 5 3 6 1 7 8)

- generates feasible offspring by identifying subsets of vertices that occupy the same subset of positions in both parents.

Mutation

Mutation involves reordering of the list:

Before:	(5 8 7 2 1 6 3 4)
After:	(5 8 6 2 1 7 3 4)

Iterated Local Search

- Problem of Local Search:
  - may get stuck in a local optimum of poor quality
  - very dependent on the starting tour
- Solution:
  - Apply Local Search to more than one tours
  - Search Diversification

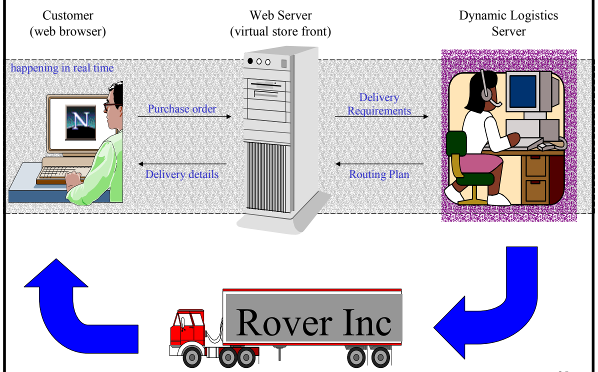
Multi-start

- **Step 1:** Create a feasible tour (randomly or by using a construction heuristic)
- **Step 2:** Apply Local Search until local optimality
- **Step 3:** Repeat from Step 1 until some stopping criteria is met
- **Step 4:** Output best tour found during this process

Multi-start: Random vs Constructive Restarts

- Random:
  - creating a random tour is fast O(n)
  - Local Search is likely to be slow
  - Local Search usually does not perform very well
- Construction Heuristics
  - creating a new tour is slow (O(n^2))
  - Local Search is usually quite fast
  - Local Search usually performs better when started from good tours

Exercise: Route Optimization



Vehicle Routing Problem with Time Windows

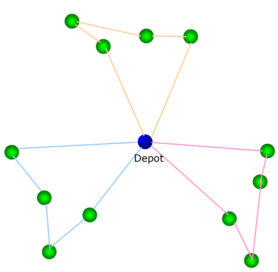
Given

- k vehicles, each with fixed capacity
- a set of customers, having location,
  time window, service duration and demand
- cost matrix $[c_{ij}]$

Find min-cost vertex-disjoint feasible routes to cover all customers.

Finding a feasible solution is NP-hard, even k=1!

Vehicle Routing Problem



Vehicle Routing Problem with Time Windows