Greedy Algorithms

Carlos Cotta

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga

http://www.lcc.uma.es/∼ccottap

Comput Eng, Softw Eng, Comput Sci & Math – 2023-2024

UNIVERSIDAD
DE MÁLAGA

Greedy Algorithms

Last Week in ADA...

Dynamic programming exploit the recurrent structure of the optimal solution to a problem to solve it implicitly exploring all possible problems.

Due to overlapping subproblems, i.e., subproblems appearing multiple times in the resolution of a problem, we can efficiently solve the latter by keeping a data structure to store their solutions.

I'm Feeling Lucky!

A dynamic programming algorithm approaches the resolution as a decision making process.

Depending upon decisions taken we are faced with different subproblems. We have to explore all possibilities and pick the one that provides the best value.

The complexity can be high due to the exhaustive exploration of all possible decisions.

It would be nice to take a single decision, without exploring any further possibility, and still find the optimum. Sometimes this is possible!

A World of Sharks

Greed, for lack of a better word, is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures, the essence of the evolutionary spirit. Greed, in all of its forms; greed for life, for money, for love, knowledge, has marked the upward surge of mankind.

Gordon Gekko (Michael Douglas)
Wall Street, 1987

The Greedy Approach

Greedy techniques are adequate to solve optimization problems.

They are a constructive approach: a solution is generated step-by-step taking decisions on how to extend it until it is complete.

In each step the decision is:

1. feasible: fulfills problem constraints.
2. locally optimal: it is the best possible decision given what it is known up to that point.
3. irreversible: once taken, a decision cannot be undone.

Required Ingredients

A greedy algorithm requires:

1. An initialization function, providing the initial state of the (empty) solution.
2. A completion function, telling whether a certain partial solution is finally completed.
3. A generation function, providing all available decisions.
4. A selection function, indicating which is the best candidate decision. This is typically related to the objective function, quantifying the goodness of the final solutions.
5. An extension function, updating the solution state given the decision taken.

A Greedy Scheme

A scheme of a greedy approach

```
proc Greedy (↓P: Problem, ↑sol: Solution, ↑feasible: 𝔹)
variables L: List⟨Decision⟩
begin
    sol ← EmptySol(P)
    feasible ← TRUE
    while ¬Complete(sol, P) ∧ feasible do
        L ← GenerateCandidates(sol, P)
        if L= ∅ then feasible ← FALSE
        else sol ← Extend(sol, P, Select(L, P))
        endif
    endwhile
end
```

Considerations

Greedy methods are:

- Simple: it is easy to conceive a greedy solution to almost any optimization problem.
- Easy to implement: typically, they do not require complex data structures.
- Efficient: their complexity is usually linear in the number of decisions, and decision generation and selection are often fast.

However not all problems can be solved to optimality using greedy methods.

Coin Change

The Coin Change problem requires a dynamic programming approach in general, but in practice a greedy approach can do.

Consider our current coins. Denominations are: 1, 2, 5, 10, 20, 50, 100, 200.

With these denominations a greedy algorithm can provide the optimal solution.

Coin Change

The decisions consist of determining which kind of coin is taken at each step.

A partial solution is a coin bag whose value is less than the target value.

The feasible candidates are those coins whose value is no larger than the difference between the current accumulated value and the target value.

Coin Change

The objective function (to be minimized) is the number of coins. Each decision increases this number by one.

The selection function tries to locally minimize the distance to the target value, taking the highest denomination available at each step.

Algorithm Pseudocode

Coin Change

```
proc CoinChange (↓M: ℕ, ↓d: ARRAY [1..n] OF ℕ,
                 ↑sol: ARRAY [1..n] OF ℕ, ↑feasible: 𝔹)
// d: denominations sorted in descending order.
   variables i: ℕ
begin
   for i ← 1 to n do sol[i] ← 0 endfor
   feasible ← TRUE; i ← 1
   while (M > 0) ∧ feasible do
      while (i≤n) ∧ (d[i]>M) do i ← i+1 endwhile
      if i>n then feasible ← FALSE
      else sol[i] ← sol[i]+1; M ← M−d[i] endif
   endwhile
end
```


Optimality Proof

Lemma 1

Let n_k be the number of coins with denomination k. In the optimal solution $n_{100} < 2$, $n_{50} < 2$, $n_{20} < 3$, $n_{10} < 2$, $n_5 < 2$, $n_2 < 3$, $n_1 < 2$. Besides, $n_{20} + n_{10} < 3$ and $n_2 + n_1 < 3$.

Proof

For 100, 50, 10, 5 and 1 cents: if we had two such identical coins, a pair could be substituted a by coin of twice its value.
For 20 cents: if we had 3 coins of this type, we could use 1 of 50 cents and 1 of 10 cents instead. Idem for 2 cents, using a coin of 5 cents and another of 1 cent.
Finally, if we had 2 coins of 20 cents and 1 of 10 cents we could use one of 50 cents instead. Idem for 2-cent and 1-cent coins with a 5-cent coin. □

Optimality Proof

Lemma 2

If a solution fulfills Lemma 1, it is unique.

Proof

Let s, s' be different solutions that fulfill Lemma 1 and yield the same total sum. Let k be the highest denomination for which $n_k > n'_k$. If the total sum is the same, s' has to use coins of denomination less than d_k to sum at least d_k. But this is impossible according to the conditions imposed by Lemma 1. Thus $s = s'$. □

Corollary 3

If a solution fulfills Lemma 1, it is optimal.

Optimality Proof

Theorem 4

The greedy approach finds the optimal solution.

Proof

The greedy approach never takes a coin with denomination k if another one $k' > k$ is feasible. Hence, it will never take more than one coin of denomination 100, 50, 10, 5, and 1, neither three coins of 20 cents or 2 cents, neither two 20-cent and one 10-cent coin or two 2-cent and one 1-cent coin. Therefore, Lemma 1 will be fulfilled and the solution will thus be optimal. □

MST Definition

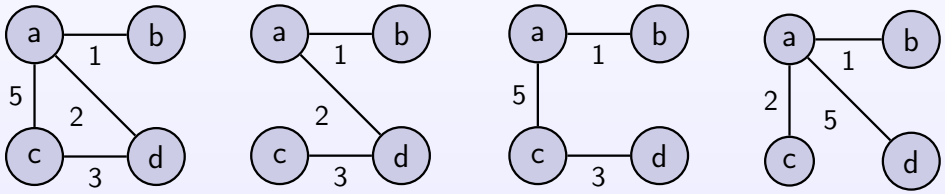
Spanning Tree

Let $G(V, E)$ be an undirected connected graph. A spanning tree for G is a tree included in G spanning all vertices in V.

Minimum Spanning Tree – MST

Let $G_W(V, E)$ be an undirected connected graph with weights w_e associated with each edge e. A minimum spanning tree of G_W is a spanning tree of G_W whose weight sum is minimal.

MST Example



Graph $w(T_1) = 6$ $w(T_2) = 9$ $w(T_3) = 8$

Searching for a MST

The number of potential spanning trees of an arbitrary graph grows superexponentially with the number of vertices.

However, the MST can be easily found using greedy techniques.

Most popular approaches are:

1. Jarník's algorithm (1930) – rediscovered by Prim (1957) and Dijkstra (1959)
2. Kruskal's algorithm (1956) – rediscovered by Loberman and Weinberger (1957)

Other approaches: Borůkva's algorithm (1926) and Chazelle's algorithm (2000), among others.

Prim's Algorithm

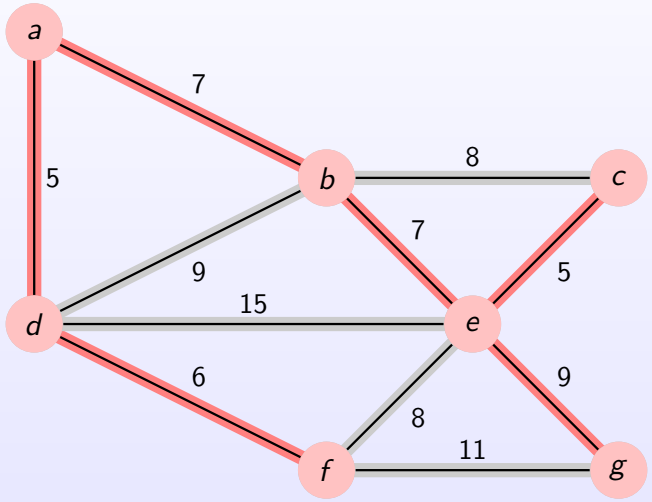
Basic Idea

Prim's algorithm builds a MST in an incremental way, expanding a subtree until it spans the whole graph:

1. Start from tree with a single (randomly chosen) node and no edges.
2. Expand the tree greedily, adding the closest vertex among those not yet chosen to the tree. the closet vertex is the one with a shortest edge to a vertex in the tree.
3. Repeat until all nodes are spanned.

Prim's Algorithm

Example



Prim's Algorithm

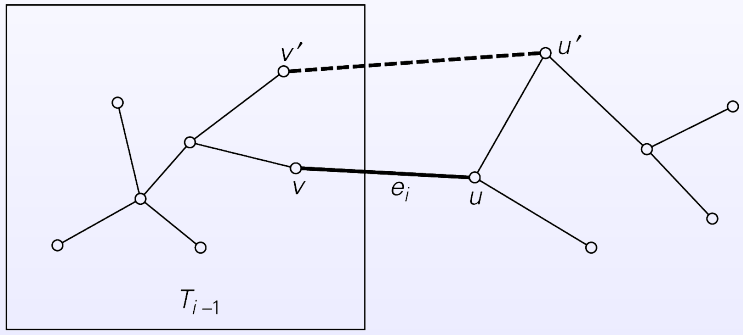
Correctness

To prove the optimality of Prim's algorithm we claim that the subtree built in each step of the algorithm is part of the MST. An inductive proof follows:

1. The initial tree T_0 has a single vertex a no edges, so it is trivially a part of the MST.
2. Assume the tree T_{i-1} at step $i-1$ ($i < n$) is included in the MST.
3. We need to prove that the edge –greedily chosen– to build T_i from T_{i-1} is also in the MST.

Prim's Algorithm

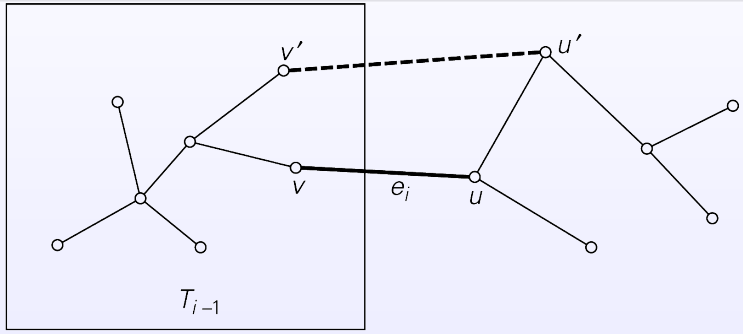
Correctness



If Prim's algorithm chose $e_i = (v, u)$ and $e_i \notin$ MST, then if we added e to the MST we would have a cycle.

Prim's Algorithm

Correctness



Let $e' = (v', u') \neq e_i$ be the edge closing the cycle. It must be $w_{e'} > w_{e_i}$ (otherwise Prim's algorithm would pick e' rather than e_i). Then, by replacing e' by e_i in the MST we would have a spanning tree of lower weight, which is impossible because we already had the MST. Hence $e_i \in$ MST.

Prim's Algorithm

Complexity

The complexity depends on the data structures used. We have to keep nodes sorted according to their distance to the tree, so we need a heap.

- Insertion, extraction and modification of an element can be done in $O(\log n)$.
- The initial construction of the heap is $O(n \log n)$ ($n-1$ insertions). Later, we make $n-1$ extractions and $|E|$ modifications.
- The overall complexity is therefore $O((n+|E|)\log n)$. Since $|E| \geqslant n$, the complexity is $O(|E| \log n)$.

Kruskal's Algorithm

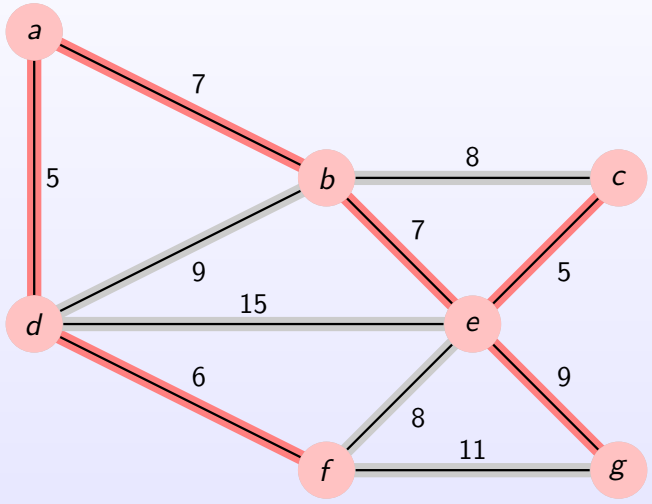
Basic Idea

Kruskal's algorithm also builds the MST incrementally and greedily, but not like Prim's: Instead of expanding a tree node by node, Kruskal's algorithm keeps a forest of trees that fuse in each step.

1. Edges are increasingly sorted according to weight.
2. n trees are created, each with a single different node of the graph and no edges.
3. The list is traversed and each edge is considered:
   - If the endpoints of the edge are in different subtrees, these are fused by using the edge.
   - If both endpoints are in the same subtree, the edge is ignored.
4. Repeat until a single edge remains.

Kruskal's Algorithm

Example



Kruskal's Algorithm

Correctness

The proof of optimality is essentially the same as for Prim's algorithm. The only (trivial) difference is that we consider acyclic unconnected graphs instead of subtrees.

Kruskal's Algorithm

Complexity

The complexity depends on the data structures used.

1. Initial edge sorting has complexity $O(|E| \log |E|)$.
2. Then we have to traverse the list, check for cycles, and fuse subtrees.

Kruskal's Algorithm

Data Structures

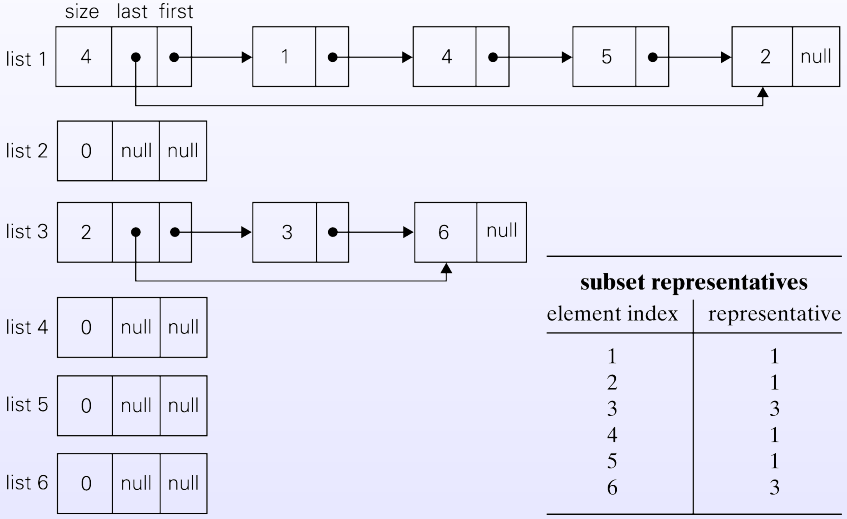
The ADT used to store the collection of subtrees has to support three operations:

- *MakeSet*(i): build a tree with just vertex i.
- *Find*(i): find the tree that contains vertex i.
- *Union*(i, j): extract from the collection the two subtrees that contain vertices i and j, fuse them and insert the result back in the collection.

We can optimize the ADT for fast search or for fast union.

Kruskal's Algorithm

Data Structures: Fast Search



subset representatives

element index	representative
1	1
2	1
3	3
4	1
5	1
6	3

Kruskal's Algorithm

Data Structures: Fast Search

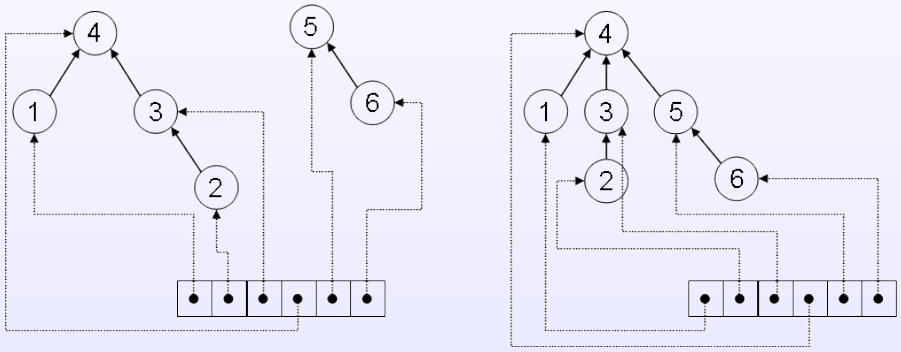
With this data structure the complexity is:

- *MakeSet*(i): build a liked list with a single element is $\Theta(1)$.
- *Find*(i): access a position of an array is $\Theta(1)$.
- *Union*(i, j): graft the shortest list into the larger and update the array. A single operation is $O(n)$, but the amortized complexity of n operations is $O(n \log n)$.

The overall complexity is that of n unions and m searches, i.e., $O(n \log n + m)$.

Kruskal's Algorithm

Data Structures: Fast Union



Kruskal's Algorithm

Data Structures: Fast Union

With this data structure the complexity is:

- *MakeSet*(i): building a tree with a single element is $\Theta(1)$.
- *Union*(i, j): linking a tree's root with another's is $\Theta(1)$.
- *Find*(i): following the pointers to the root is $O(n)$. If we always fuse the the shallowest tree into the deepest one, the height of the trees –and hence the complexity of getting to the root– is $O(\log n)$.

The overall complexity is that of n unions and m searches, i.e., $O(n + m \log n)$.

Single-Source Shortest Paths

Given a connected digraph with weights $G_W(V, E)$ and a vertex $s \in V$, the Single-Source Shortest-Paths Problem (SSSP) consists of finding the shortest paths from s to each of the remaining vertices.

As long as all weights are non-negative, a greedy approach discovered by E. Dijkstra in the 1950s can provide the optimal solution.

Dijkstra's Algorithm

Basic Idea

Dijkstra's algorithm works analogously to Prim's algorithm.

Vertices are split in two groups: explored (those for which the optimal path is already known) and unexplored (for which the optimal path is still to be found).

The next explored vertex is greedily selected.

Dijkstra's algorithm finds the shortest path to each vertex in order of distance: in the i-th step the shortest paths to i vertices are known.

Dijkstra's Algorithm

Basic Idea

More precisely:

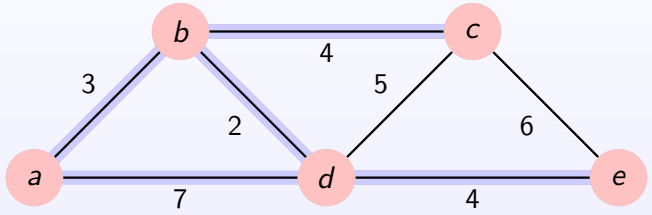
1. Let S be the set of explored vertices. Initially $S = \{s\}$. Let d_u be the optimal distance from s to $u \in S$. Initially $d_s = 0$.
2. The closest vertex not in S is sought, i.e., the one minimizing

$$\pi(u) = \min_{(v,u)\in E, v\in S} \left(w_{(v,u)} + d_v\right)$$

3. u is added to S, and we update $d_u = \pi(u)$. We repeat this until $S = V$.

Dijkstra's Algorithm

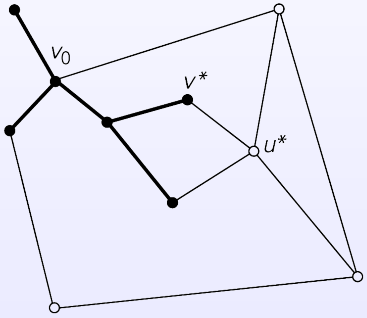
Example



S	$v \mid (d_v + w_{uv})$			
	b	c	d	e
$\{a\}$	$a \mid (0+3)$	$- \mid \infty$	$a \mid (0+7)$	$- \mid \infty$
$\{a, b\}$	$a \mid 3$	$b \mid (3+4)$	$b \mid (3+2)$	$- \mid \infty$
$\{a, b, d\}$	$a \mid 3$	$b \mid (3+4)$	$b \mid 5$	$d \mid (5+4)$
$\{a, b, d, c\}$	$a \mid 3$	$b \mid 7$	$b \mid 5$	$d \mid (5+4)$
$\{a, b, d, c, e\}$	$a \mid 3$	$b \mid 7$	$b \mid 5$	$d \mid 9$

Dijkstra's Algorithm

Correctness



If the optimal paths from s to the remaining $n-1$ nodes are sorted, the i-th path is discovered at step i. This can be inductively proved:

1. In the first step we discover the shortest direct path.
2. Assume that in the $(i-1)$-th step we discovered the $i-1$ shortest paths.
3. The i-th vertex u^* is the closest to s via explored nodes. Any other vertex is farther, so there cannot be a shorter path to u^*.

Dijkstra's Algorithm

Complexity

As in Prim's algorithm, we can use a heap to sort nodes according to their distance to s via explored nodes.

The complexity is $O(|E| \log n)$. If we want to find all shortest paths in the graph, the process can be repeated from n different initial nodes. This is $O(|E| n \log n)$. If the graph is not very dense, this is better than Floyd-Warshall algorithm $O(n^3)$.

Using more sophisticated data structures (Fibonacci heaps), the computational cost of the SSSP problem is $O(n \log n + |E|)$, and the all-shortest-paths problem is $O(n^2 \log n + |E| n)$.

You can't always win...

In general most problems cannot be solved using greedy approaches.

Despite this, greedy techniques can be worth as an approximate or heuristic approach.

Recall...

1. An approximate approach does not ensure finding the optimal solution but only a solution within certain distance to the optimum.
2. A heuristic approach does not ensure any closeness to the optimum, but often (if well-designed) they provide optimal or near-optimal solutions.

Example: The Knapsack Problem

Given a knapsack of capacity W and n objects with weights and values (p_i, v_i), we have to select objects in a greedy way.

Two naïve approaches:

1. Take in each step the most valuable object fitting in the knapsack until no one is left.
2. Take in each step the lightest object fitting in the knapsack until no one is left.

Example: The Knapsack Problem

Consider this problem instance (W=5 kg):

object	weight	value
1	2 kg	12€
2	1 kg	10€
3	3 kg	20€
4	2 kg	15€
5	1 kg	5€

According to the first heuristic:

1. Pick $o_3 \Rightarrow V = 20€,\ W = 2$.
2. Pick $o_4 \Rightarrow V = 35€,\ W = 0$.

According to the second heuristic:

1. Pick $o_2 \Rightarrow V = 10€,\ W = 4$.
2. Pick $o_5 \Rightarrow V = 15€,\ W = 3$.
3. Pick $o_4 \Rightarrow V = 30€,\ W = 1$.

The optimal solution (computed by dynamic programming) is 37€.

The Fractional Knapsack

Consider a variant of the problem in which a fraction of an object can be put in the knapsack, i.e., solutions are vectors $\langle x_1, \cdots, x_n \rangle$, where $x_i \in [0, 1]$.

Lemma 5

The fractional knapsack problem exhibits optimal substructure.

Proof

The proof is the same we saw for the 0-1 knapsack problem. □

The Fractional Knapsack

Theorem 6

Let o_i the object with the largest ratio $\delta_i = v_i/p_i$. The optimal solution includes the maximum possible fraction of that object.

Proof

By *reductio ad absurdum*: let $x_i < 1$ and let $x_j > 0$ $(i \neq j)$ in the optimal solution. Object o_j uses $x_j p_j$ space in the knapsack and contributes $V = x_j v_j = x_j p_j \delta_j$ euros. If we change that fraction of o_j by a fraction of o_i of the same weight we have $V' = x_j p_j \delta_i > V$, which is a contradiction. □

The Fractional Knapsack

Corollary 6

The greedy algorithm that sorts objects by decreasing values of value/weight and considers them in order until completing the capacity of the knapsack is optimal.

Consider this problem instance (W=5 kg):

object	weight	value	
1	1 kg	6€	6 €/kg
2	2 kg	10€	5 €/kg
3	3 kg	12€	4 €/kg

The heuristic order is o_1, o_2, o_3:

1. Pick $o_1 \Rightarrow V = 6€$, $W = 4$.
2. Pick $o_2 \Rightarrow V = 16€$, $W = 2$.
3. Pick 2/3 of $o_3 \Rightarrow V = 24€$, $W = 0$.

As for the 0-1 Knapsack...

If the previous instance was discrete rather than fractional, the heuristic value would be 16€ whereas the optimum ($\{o_2, o_3\}$) has a value of 22€.

For the previous example (W=5 kg):

object	weight	value	
1	2 kg	12€	6.00€/kg
2	1 kg	10€	10.00€/kg
3	3 kg	20€	6.33€/kg
4	2 kg	15€	7.50€/kg
5	1 kg	5€	5.00€/kg

The heuristic order is o_2, o_4, o_3, o_1, o_5, and the resulting solution is $\{o_2, o_4, o_1\}$ with a value of 37€.

The Knapsack Problem

In general the greedy heuristic will not provide the optimal solution to the discrete version.

We do not have any guarantee of closeness to the optimum either.

Despite this, it remains a very popular approach to solve the problem in its multiple variants, e.g., the multidimensional 0-1 knapsack.

Task Scheduling in Parallel Machines

Parallel Machine Scheduling

We have m identical machines and n tasks to be run on any of them. Let T_i the runtime for the i-th task.

We wish to find an assignment of tasks to machines that minimizes the total runtime for the n tasks.

We look for an appropriate distribution of the workload among tasks. This is called load balancing.

Load Balancing

Let $A_{1\times n}$ be the task assignment, i.e., $A_i = k$ means that the i-th task is run on the k-th machine.

The total runtime for the k-th machine is

$$T_k^{\Sigma} = \sum_{A_i=k} T_i \ .$$

Likewise, the total system runtime is

$$T^{\Sigma} = \max_{1\leqslant k\leqslant m} T_k^{\Sigma} \ .$$

A greedy approach tries to minimize runtime by assigning the next tasks to the first machine that becomes available.

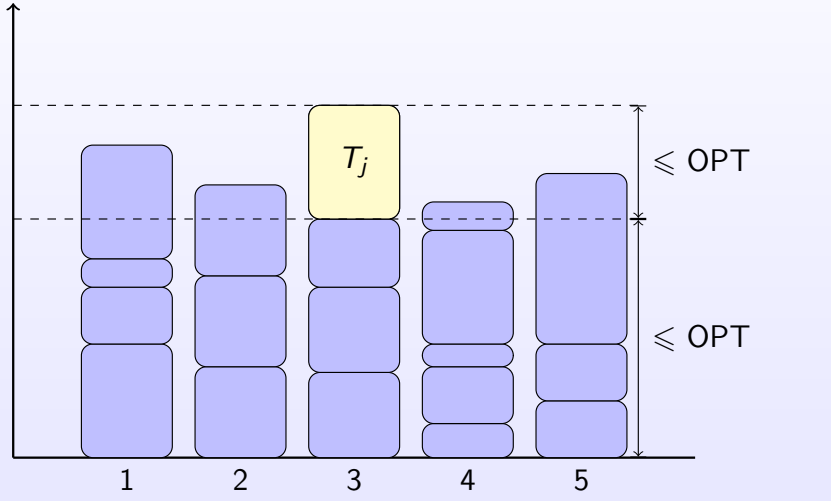
A Load Balancing Heuristic

Load Balancing

```
proc Balanced (↓T: ARRAY [1..n] OF ℕ, ↓m: ℕ,
               ↑A: ARRAY [1..n] OF ℕ)
variables i, j, min:ℕ;  Tot: ARRAY [1..m] OF ℕ
begin
   for i ← 1 to m do Tot[i] ← 0 endfor
   for i ← 1 to n do
      min ← 1
      for j ← 2 to m do
         if Tot[j]<Tot[min] then min ← j endif
      endfor
      A[i] ← min; Tot[min] ← Tot[min]+T[i]
   endfor
end
```


Load Balancing at Work

analysis proceed



Analysis of the Heuristic

Let OPT be the optimal runtime of the n tasks in the m machines. We can bound this quantity as:

1. $OPT \geqslant \max_{1 \leqslant i \leqslant n} T_i$
2. $OPT \geqslant \frac{1}{m} \sum_{1 \leqslant i \leqslant n} T_i$

Let T_j be the last task to finish and let k be the corresponding machine. According to (1) above, $T_j \leqslant OPT$.

Analysis of the Heuristic

Consider $T_k^{\Sigma} - T_j$:

1. The greedy assignment policy implies that at the time s in which task j is assigned, $T_k^{\Sigma[s]} \leqslant T_{k'}^{\Sigma[s]}$, $(1 \leqslant k' \leqslant m)$.
2. It follows that

$$m T_k^{\Sigma[s]} \leqslant \sum_{1 \leqslant k' \leqslant m} T_{k'}^{\Sigma[s]} \leqslant \sum_{1 \leqslant i \leqslant n} T_i$$

and hence $T_k^{\Sigma[s]}$ is less than the average runtime of all machines.

Therefore, $T_k^{\Sigma[s]} = T_k^{\Sigma} - T_j \leqslant OPT$ according to (2).

Since $T^{\Sigma} = T_k^{\Sigma[s]} + T_j$, the greedy solution is at most twice the optimal solution.

back to the example

The Travelling Salesman Problem

Travelling Salesman Problem (TSP)

We have n cities and a matrix $D_{n \times n} = \{d_{ij}\}$ where d_{ij} is the distance between city i and city j.

We look for a minimum-length Hamiltonian tour (a path that goes through each city exactly once and returns to the original city).

This is a classical *NP−hard* problem. No general efficient method for finding the optimal solution is known. The same applies to approximation schemes.

Heuristics are the only practical method to deal with large instances of problems like this.

Heuristics for the TSP

Notice the similarity between the optimal solution to the TSP and a MST:

1. all vertices must be spanned.
2. the total weight must be minimized.

We can use heuristics similar to the methods known for finding MSTs. These must be adapted so that each node has exactly two neighbors.

Heuristics for the TSP

Consider these two heuristics:

1. Nearest Neighbor: start at any city and go in each step to the closest non-yet-visited city.
2. Path Merging: consider edges by increasing weight and add them to the solution if no constrain is violated.

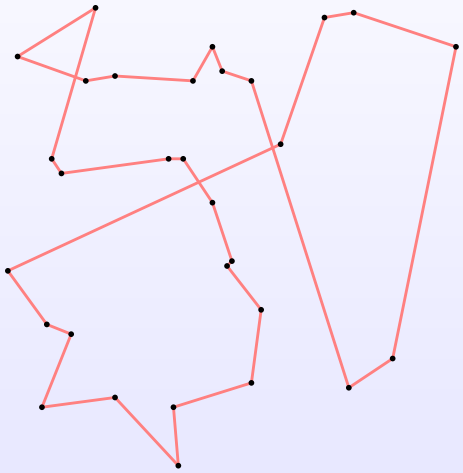
Nearest Neighbor Heuristic

Example

Consider this 30-city TSP instance and apply NNH from a certain starting city.

The final cost is 546.

next



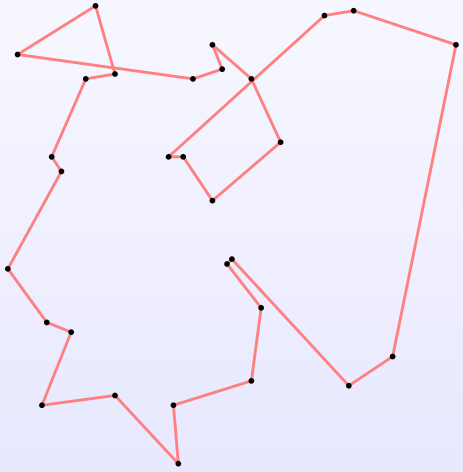
Nearest Neighbor Heuristic

Example

If the process is repeated from a different starting city, the result is different.

The final cost is 498.

previous next



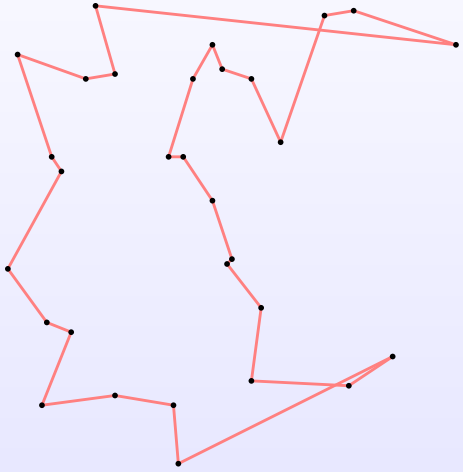
Path Merging Heuristic

Example

Consider now the path merging heuristic.

The final cost is 474.

previous



Complementary Bibliography


R.L. Graham and P. Hell
On the History of the Minimum Spanning Tree Problem
Annals of the History of Computing 7(1):43–57, 1985
https://www2.seas.gwu.edu/~simhaweb/champalg/mst/papers/GrahamMSTHistory.pdf

Image Credits

- Picture of Michael Douglas as Gordon Gekko: snapshot from *Wall Street* (directed by Oliver Stone, 1987, © 20th Century Fox)
- Euro coins: public domain
- Prim's algorithm and Dijkstra's algorithm correctness, Kruskal's algorithm fast search: A. Levitin, © 2007 Pearson Addison-Wesley
- Picture of Edsger Dijkstra: Hamilton Richards, CC-BY-SA 3.0