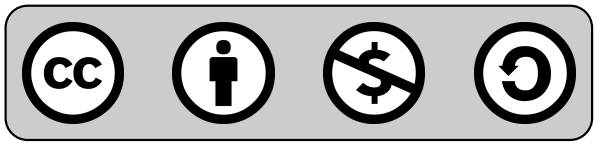
FSD301 Optimization in graphs

Christophe Garion
ISAE-SUPAERO – DISC

License CC BY-NC-SA 3.0



Course presentation

Main **learning outcomes** (at the end of the lecture you should be able to...):

- model problems using graphs and use some state-of-the-art algorithms to solve them
- understand how to prove correctness of algorithms
- use a Python library to implement graphs and use algorithms
- implement and test a complex algorithm on graphs

Course syllabus

Schedule:

September, 9th 2020 13:45	definitions, implementation of graphs, graph traversal, shortest path
September, 15th 2020 13:45	lab session on networkx and shortest paths
September, 21st 2020 09:00	MST and TSP
September, 21st 2020 13:45	evaluated lab

The team:

- Christophe Garion (ISAE-SUPAERO/DISC)
- Xavier Olive (ONERA/DTIM)
- Olivier Poitou (ONERA/DTIM)
- Xavier Thirioux (ISAE-SUPAERO/DISC)

Aknowledgements

The following online courses have been a great inspiration for this course:

Roughgarden, Tim (2018).
Tim Roughgarden's Online Courses on Algorithms Analysis and Design.
Stanford Theory Group.
http://theory.stanford.edu/~tim/videos.html.

Do not hesitate to attend them!

Outline

Outline

What is a graph?

Definition (graph)

A graph is a pair $\langle V, E \rangle$ where:

- V is a set of **vertices**, or **nodes**
- E is a binary relation on V called the **adjacency relation**

There is no particular constraint imposed on the relation E.

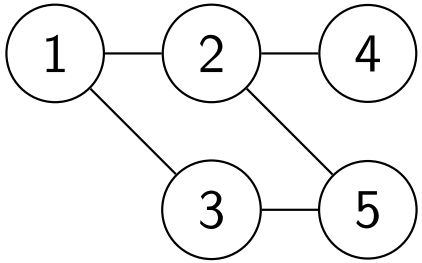
If the pairs of E are unordered, the graph is **undirected**, else it is **directed** (also called a **digraph**).

Intuitively, if you use a graph to model connections between towns and $\langle \text{Toulouse}, \text{Ramonville} \rangle \in E$, it means:

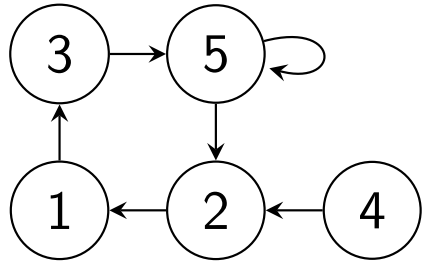
- that you can go from Toulouse to Ramonville and from Ramonville to Toulouse in an undirected graph.
- that you can go from Toulouse to Ramonville, but not necessary from Ramonville to Toulouse in a directed graph.

Graphical representation and vocabulary

Undirected graph:



Directed graph:



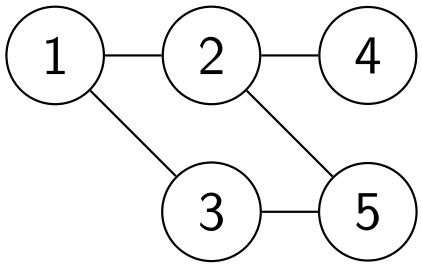
In the **directed** graph:

- edges are also called **arrows**
- (5) is called the **source**, the **tail** or the **origin** of the edge going to (2)
- (2) is called the **destination** or the **head** of the edge coming from (5)
- there is a **loop** on (5)

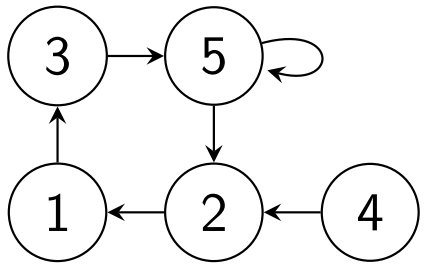
We will not consider in this lecture multiple graphs, i.e. graphs in which several edges can go from the same source to the same destination.

Graphical representation and vocabulary

Undirected graph:



Directed graph:

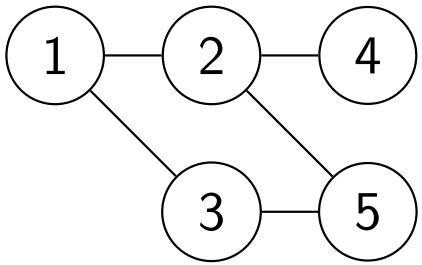


Some measures:

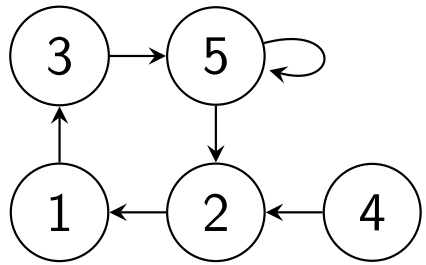
- the **order** is $|V|$ (5 in the graphs)
- the **size** is $|E|$ (10 for the undirected graph, 6 for the directed graph)
- the **degree** of a vertex $v \in V$ is $|\{v' | (v, v') \in E\}|$ (2 for node (1) for instance)
  For a directed graph, **indegree** and **outdegree** are distinguished.

Graphical representation and vocabulary

Undirected graph:



Directed graph:



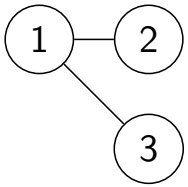
There is a **path** between the vertices v_s and v_d of G iff there is a **finite** sequence of vertices $v_0, ..., v_n$ such that

- $v_0 = v_s$
- $v_n = v_d$
- $\forall i \in \{0, \ldots, n-1\}, (v_i, v_{i+1}) \in E$

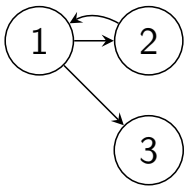
A path such that $v_s = v_d$ is called a **cycle**, e.g. 1-3-5-2-1 in the directed graph.

Where are the graphs?

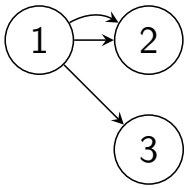
Find the graphs we will consider in the lecture among these representations.



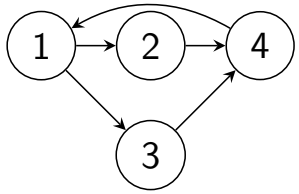
☐



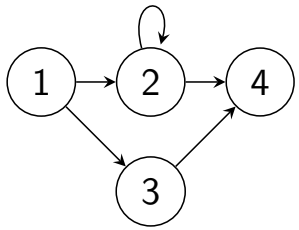
☐



☐



☐



☐

How many edges?

Exercise

Considering a graph $\mathcal{G}$ with n nodes, what is the maximum number of edges if:

- $\mathcal{G}$ is undirected
- $\mathcal{G}$ is directed

Outline

Implementating graph as matrix

E is a binary relation:

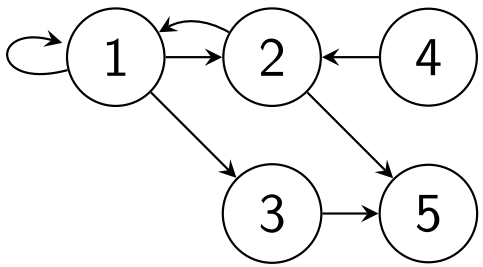
- encode E as a $|V| \times |V|$ matrix $\mathcal{M}$
- injection $nti : V \rightarrow [1, |V|]$

$$\forall (v_s, v_d) \in V^2 \ \mathcal{M}_{nti(v_s), nti(v_d)} = \begin{cases} 0 \text{ if } (v_s, v_d) \notin E \\ 1 \text{ if } (v_s, v_d) \in E \end{cases}$$

An example of adjacency matrix

Exercise

Consider the following graph. Can you write an adjacency matrix for it?

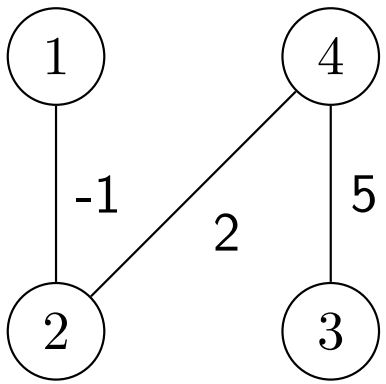


Beware with labels!

Labels attached to edges can be used as values to represent distances for instance.

But you have to define a special value to represent the non existence of edges. This value should not be in the domain of labels!

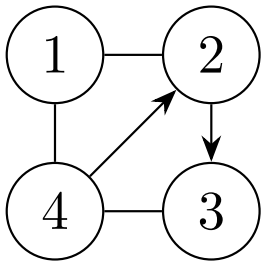
For instance:



$$\begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 5 \\ 0 & 2 & 5 & 0 \end{bmatrix}$$

Implementing edges as adjacency lists

For each node, use a list for its neighbors:



nodes	adjacency list
1	{2, 4}
2	{1, 3}
3	{4}
4	{1, 2 ,3}

Easy to use in Object-Oriented programming for instance.

The complexity point of view

Exercise

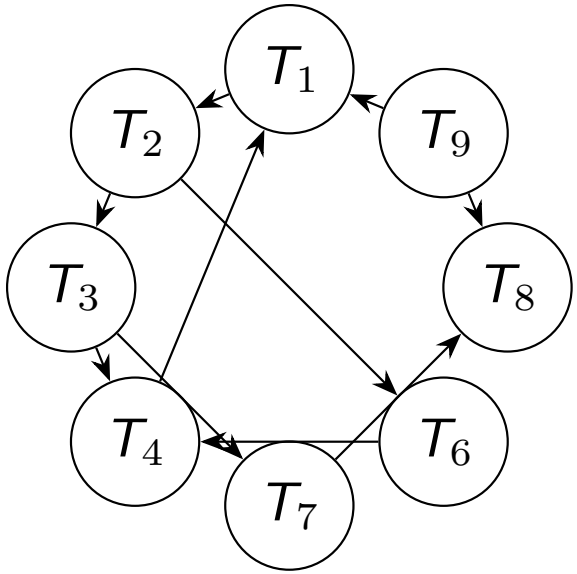
Can you complete the following table with O complexity bounds?

	adj. list	adj. matrix
add a vertex		
add an edge		
find an edge		
space to store graph		

Outline

Why searching?

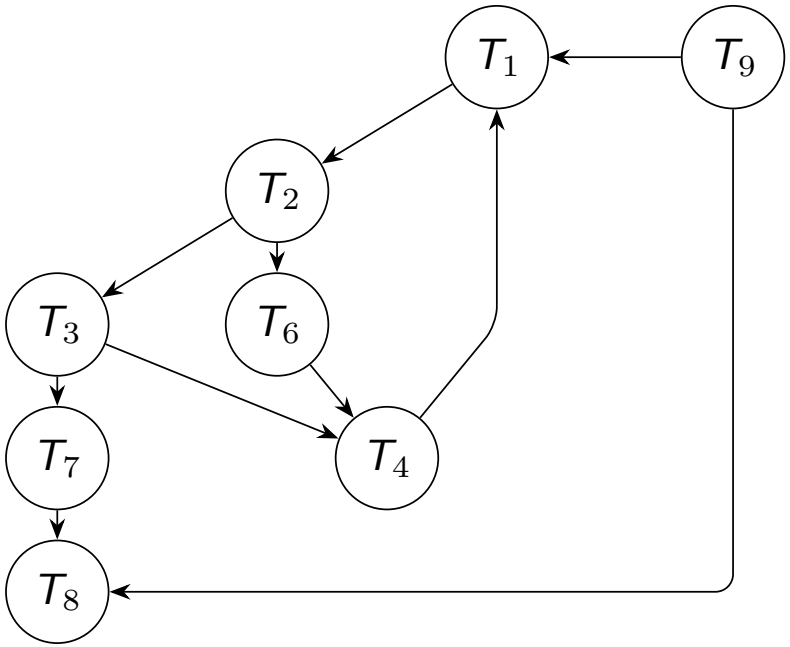
Imagine that you model the dependencies between the tasks assigned to an average SUPAERO student:



You want to know what to do before beginning task T_1. How can you do that?

Why searching?

Imagine that you model the dependencies between the tasks assigned to an average SUPAERO student:

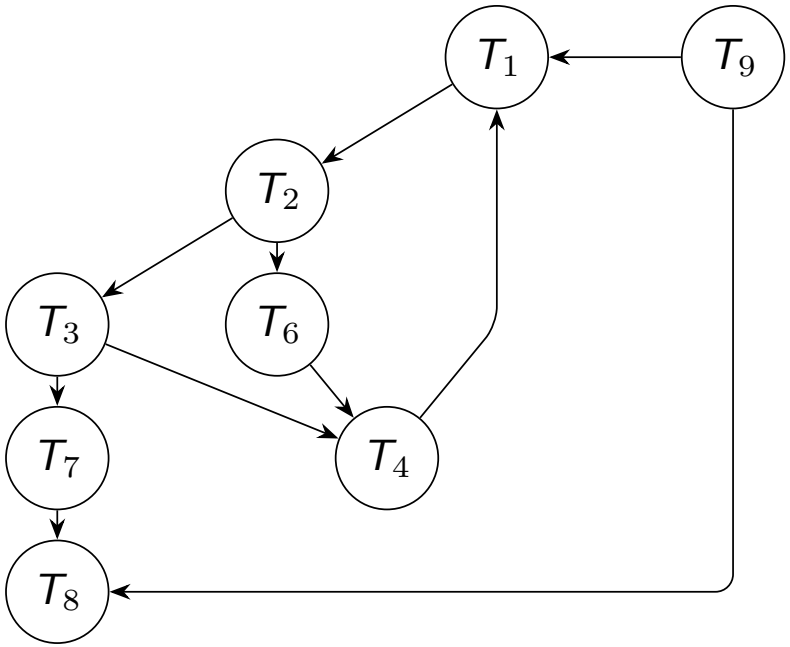


You want to know what to do before beginning task T_1. How can you do that?

Of course, you may begin by redrawing the graph...

Why searching?

Imagine that you model the dependencies between the tasks assigned to an average SUPAERO student:



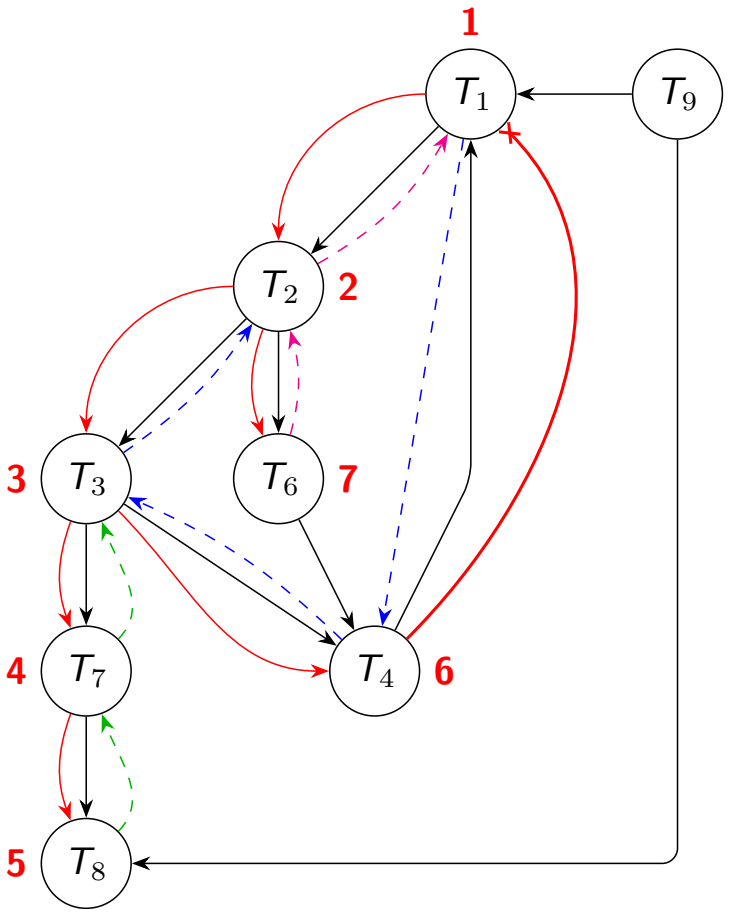
You want to know what to do before beginning task T_1. How can you do that?

Finding which vertices are reachable from T_1 is called graph searching or graph traversal.

Depth-first search

First traversal method: depth-first search (DFS)

Intuition: when discovering a new node, explore its children before exploring its siblings.



Implement DFS

Exercise

Can you write a recursive implementation of DFS? An imperative one?

Correctness

Property

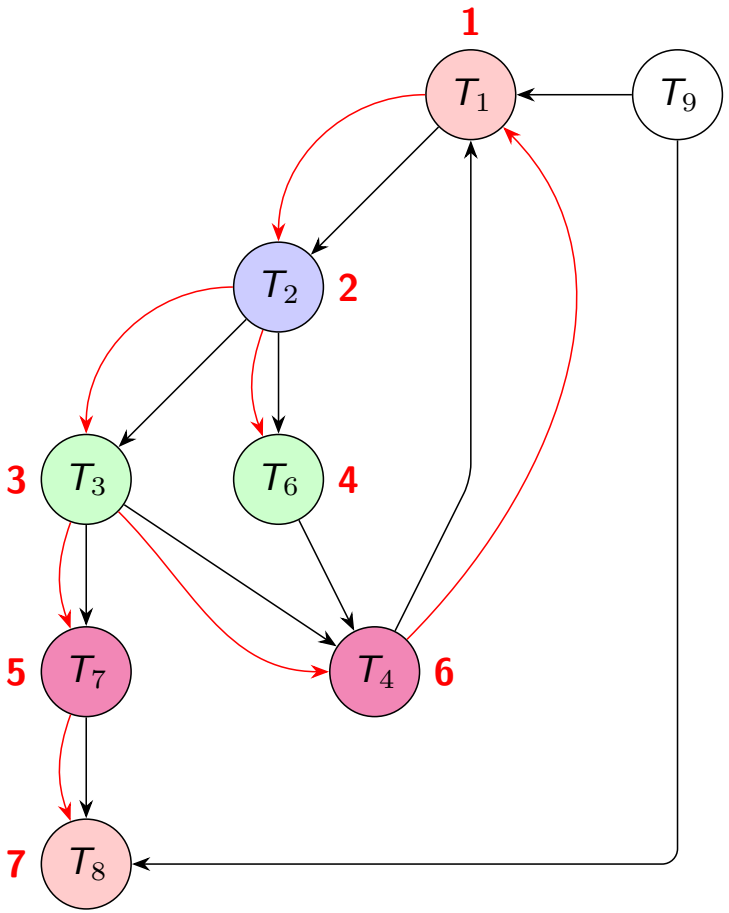
DFS starting from a vertex v will discover all nodes reachable from v.

Easy to prove using proof by contradiction.

Breadth-first search

Second traversal method: breadth-first search (BFS)

Intuition: when a node is discovered, explore its siblings before exploring its children.



Implement BFS

Exercise

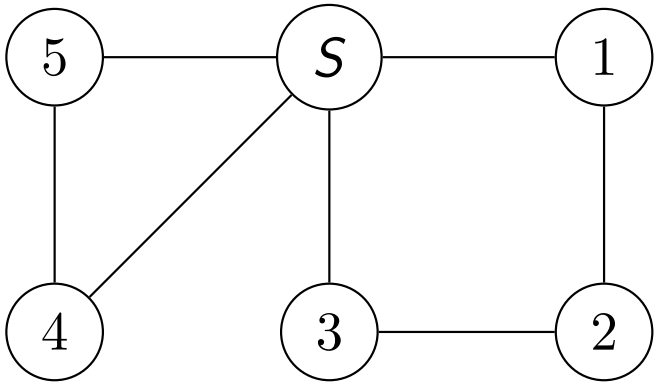
Can you write an imperative implementation of BFS?

Outline

Outline

Shortest paths: a simple example

Consider the following graph:



What are the shortest paths from (S) to the other nodes?

Shortest path: already seen...

Exercise

Write an algorithm to compute shortest path from (S) to the other nodes.

Now, it has to be proven

Exercise

How can you prove that the BFS computes the shortest path?

Two parts:

1. prove the `dist[v]` is the correct distance
2. prove the path obtained by `pred[v]` is the shortest

As the path computed by `pred[v]` has length `dist[v]`, we only have to prove the first assertion.

Complexity of BFS

Exercise

What is the complexity of BFS?

Outline

And now, with lengths!

Ok, but when you want to compute shortest paths on a map, you have **distances on edges**.
So, we should put distances, **weights** or **costs** on the edges.

Can you use BFS for such graphs?

Yes, if the costs are natural numbers, you can add artificial nodes to use a BFS.

➥ what do you think of this solution?

Dijkstra's algorithm

Edsger W. Dijkstra (1930–2002) is one of the most influential computer scientist of 20th century. He received the 1972 Turing Award and has written papers among the most important for CS.

Dijkstra has discovered an algorithm to compute shortest paths from a **single node** to all **reachable** nodes in a graph if the weights on the edges are **positive**.

He is also famous for some "angry" quotes:

> *Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians.*
>
> *We must give industry not what it wants, but what it needs.*

☺

Dijkstra's algorithm

Procedure SPD(G, S): Dijkstra's shortest path algorithm

Input: a graph G with edges labelled with positive numbers representing distances between vertices and a starting vertex S

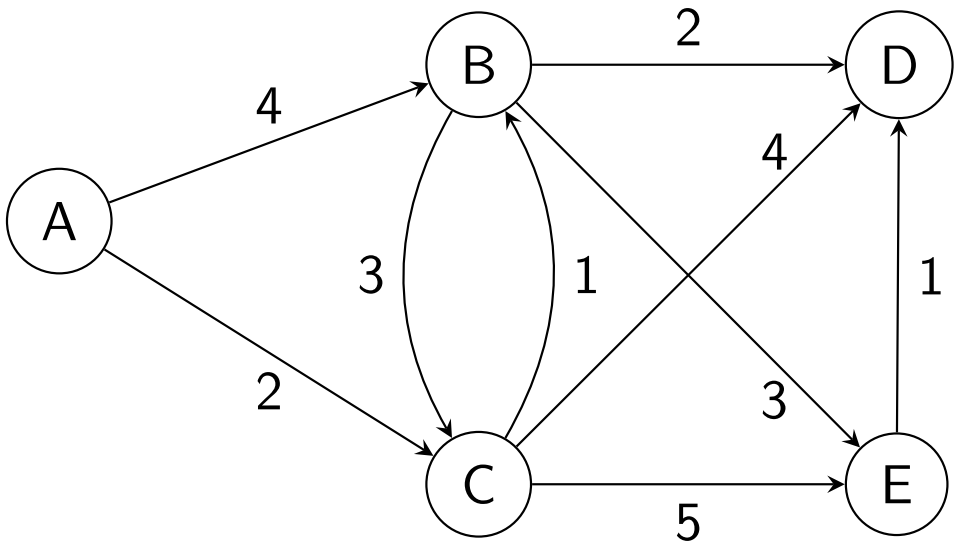
Result: at the end of the algorithm, the nodes reachable from S are known as well as their predecessor on the shortest path from S and the length of such shortest path

```
1  create an empty set N;
2  foreach vertex v of G do
3  |  dist[v] = ∞ ;
4  |  pred[v] = null_node ;
5  |  add v in N;
6  end
7  dist[S] = 0;
8  while N is not empty do
9  |  v = the node in N with the minimal distance ;
10 |  remove v from N;
11 |  foreach neighbor n of v do
12 |  |  if n ∈ N then
13 |  |  |  d = dist[v] + weight of (v, n) ;
14 |  |  |  if d < dist[n] then
15 |  |  |  |  dist[n] = d;
16 |  |  |  |  pred[n] = v;
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

Dijkstra: an example!

Exercise

Apply Dijkstra's algorithm on the following graph starting from (A).



Correctness of Dijkstra's algorithm

Exercise

Can you prove the correctness of Dijkstra's algorithm?

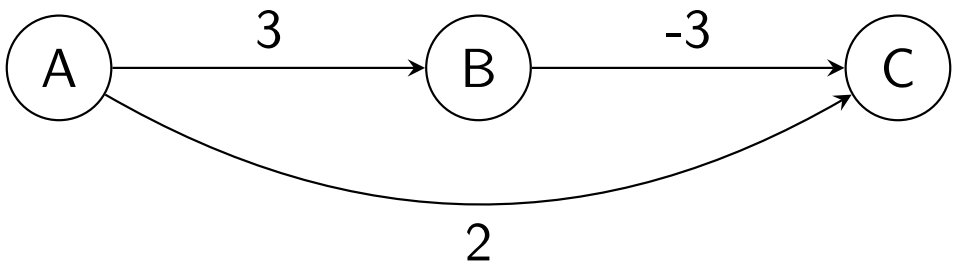
Complexity

Exercise

What is the complexity of Dijsktra's algorithm?

Using negative weights

Dijkstra's algorithm does not work with negative weights. Try it on the following example, you can also try to add some value to all edges to have positive weights.



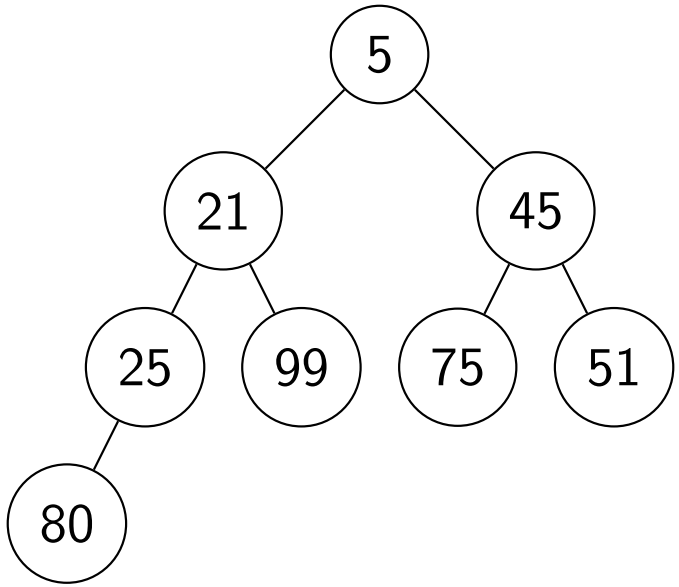
You can use for instance Floyd-Warshall algorithm to deal with negative weights.

Some remarks on implementation

The bottleneck of the algorithm is the choice of the undiscovered vertex with minimum distance. Using linear search is of course not the best solution, can we do better?

We can use heaps to implement an efficient data structure for this purpose.

Heaps are complete binary trees with the following property: the value associated to a given node is always less (or greater) than the value of its children:



Some remarks on implementation

The bottleneck of the algorithm is the choice of the undiscovered vertex with minimum distance. Using linear search is of course not the best solution, can we do better?

We can use heaps to implement an efficient data structure for this purpose.

There are lots of implementation of heaps, among which:

- binary heaps, the classical implementation that can use an array
- Fibonacci heaps, for which the complexity of the algorithm is $O(|E| + |V| \log |V|)$

Beware, you are to be able to "locate" graph nodes in the heap as distance is updated during the algorithm execution. Hash tables can be used.

Link with dynamic programming

Dynamic programming is a mathematical optimization method developed by Richard Bellman.

It is based on Bellman's Principle of Optimality:

Principle

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

Idea: break problem in simpler sub-problems recursively.

From Dijkstra's seminal paper:

Problem 2. Find the path of minimum total length between two given nodes P and Q.
We use the fact that, if R is a node on the minimal path from P to Q, knowledge of the latter implies the knowledge of the minimal path from P to R.

Dynamic programming and divide and conquer

So, it seems that dynamic programming is simply divide and conquer.

Not really, as dynamic programming implies the two properties for the considered problem:

- optimal substructure: the solution of the problem can be built from the optimal solutions of its subproblems
- overlapping subproblems: the subproblems are reused several times (think of Fibonacci!)

The second point explains why dynamic programming cannot be used for Quicksort.

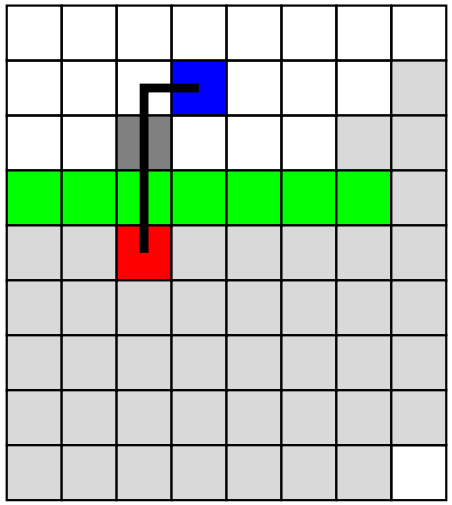
Dynamic programming can be implemented in two ways:

- top-down: classic recursion with memoization
- bottom-up: compute first subproblems and generate solutions for bigger and bigger problems

Outline

Can we beat Dijkstra?

Let us consider the following map:



We want to find the shortest path from  to  knowing that going to or exiting a  cell has a cost of 4.

Dijkstra's algorithm will develop all  nodes before considering node  which is part of the actual shortest path.

But should it not be a good idea to try to expand **first** nodes that are **closer to the goal**?

Greedy best first search

Lazy professor warning: to illustrate algorithms, we will use Red Blob Games excellent pages on algorithms applied to games. Do not hesitate to read them!

Choosing which node to consider in Dijksta's algorithm is done by removing from the **priority queue** the node with the **smallest actual distance** from  .

What happens if we choose an **estimated** distance to  in the priority queue instead?

➥ we obtain greedy best first search algorithm

In our previous problem, using Manhattan distance as an approximation of the real distance to  works!

Unfortunately, greedy best first search is not an exact method, it may return a suboptimal path!

The A* algorithm

The idea of the A* algorithm is to **combine** both **actual** distance from  and **estimated** distance to  .

The cost function used in the priority queue is now $a(n) + h(n)$ where

- $a(n)$ is the actual distance from  to node n
- $h(n)$ is the estimated distance from n to  (Manhattan distance here)

And **it works perfectly** (see demo)!

But does it always work?

Good heuristic functions

The correctness of the A* algorithm depends on the heuristic function h chosen to estimate the distance to the goal.

Definition (heuristic)

Let $h(n)$ be the estimated distance from n to the goal and $h^*(n)$ be the real distance from n to the goal.

- h is **perfect** if $h = h^*$
- h is **admissible** if $\forall n \; h(n) \leq h^*(n)$
- h is **monotonic** if $\forall n, n' \; h(n) \leq h(n') + w(n, n')$ where $w(n, n')$ is the weight of the edge between n and n' (triangle inequality)

Good heuristic functions

The correctness of the A* algorithm depends on the heuristic function h chosen to estimate the distance to the goal.

Theorem (correctness of A*)

- *if h is perfect, then A* converges immediately to the goal*
- *if h is admissible, then A* is guaranteed to find a shortest path*
- *if h is monotonic, then h is admissible and for every developed node, the computed path from the origin is optimal (no need to consider a node several times)*

Good heuristic functions

The correctness of the A* algorithm depends on the heuristic function h chosen to estimate the distance to the goal.

Theorem (complexity of A*)

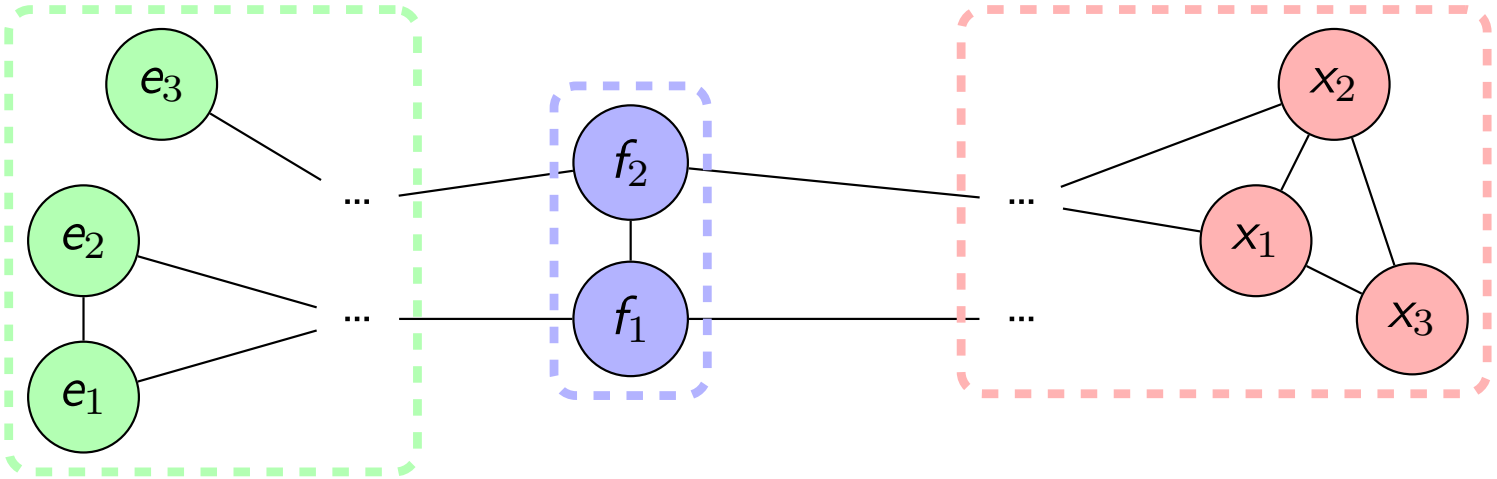
- *if h is perfect, then A* converges immediately to the goal*
- *if h is admissible, no optimal algorithm using the same h as A* will expand fewer nodes than A**
- *if h is monotonic, then complexity is linear*
- *the further h is from h^*, the greater the number of expanded nodes is*

Implementation details

- data structures are important, like in Dijkstra's algorithm: binary heap, Fibonacci heap, hash tables
- beware of heuristic function: h and a must have the same "unit".

  A classic error: if you use euclidean distance, then using `sqrt` takes time to compute. But you cannot consider $(x_j - x_i)^2 + (y_j - y_i)^2$ instead of $\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$ for h as you may overestimate your distance to goal!

Summary on shortest paths



E (explored)	F (frontier)	X (not explored)
`dist[v]` $< \infty$	`dist[v]` $< \infty$	`dist[v]` $= \infty$
minimum	minimum for paths in E finishing in F	
v $\notin$ treatment queue	v $\in$ treatment queue	v $\in$ treatment queue

Invariants

neighbors(E) $\subseteq$ E $\cup$ F
neighbors(X) $\subseteq$ F $\cup$ X

Credits

- portrait of Edsger W. Dijkstra: https://en.wikipedia.org/wiki/Edsger_W._Dijkstra#/media/File:Edsger_Wybe_Dijkstra.jpg, Hamilton Richards, CC BY-SA 3.0

Outline

Introductory problem

From the USA Computing Olympiad:

Exercise (Water flooding)

In a city there are n houses, each of which is in need of a water supply. It costs w_i dollars to build a well at house i, and it costs $c_{i,j}$ to build a pipe between houses i and j. A house can receive water if either there is a well built there or there is a path of pipes from this house to a house with a well.
You want to find the minimum amount of money needed to supply every house with water.
How can you model this problem with a graph? What is the solution to this problem?

Outline

Prim and Kruskal algorithms in 180s

Prim and Kruskal algorithms are **greedy** algorithms and **they work perfectly**!

- Prim's algorithm (1957): start from a node s and add at each step the node that can be attached as cheaply as possible to the tree already built
- Kruskal's algorithm (1956): insert edges from the graph by increasing cost. An edge is inserted as long as it does not create a cycle when added

Prim and Kruskal algorithms in 180s

Prim and Kruskal algorithms are **greedy** algorithms and **they work perfectly**!

- Prim's algorithm (1957): start from a node s and add at each step the node that can be attached as cheaply as possible to the tree already built
- Kruskal's algorithm (1956): insert edges from the graph by increasing cost. An edge is inserted as long as it does not create a cycle when added

Prim and Kruskal algorithms in 180s

Prim and Kruskal algorithms are **greedy** algorithms and **they work perfectly**!

- Prim's algorithm (1957): start from a node s and add at each step the node that can be attached as cheaply as possible to the tree already built
- Kruskal's algorithm (1956): insert edges from the graph by increasing cost. An edge is inserted as long as it does not create a cycle when added

Try them on the previous example.

Proving the algorithms: the cut property

In order to prove these algorithms, we need the following property.

Theorem (cut property)

Let $G = \langle V, E \rangle$ a graph such that all edge costs are distinct. Let S be any subset of nodes that is neither empty nor equal to V, and let $e = (v, w)$ be the minimum cost edge with one end in S and the other in $V - S$. Then every minimum spanning tree on G contains e.

How can we prove it?

direct proof	□
proof by contradiction	□
proof by recurrence	□

Proof of the cut property

Let us suppose that T is a spanning tree that does not contain e.

Idea: find an edge e' in T that can be exchanged with e.

➥ draw the graph...

Proof of Prim's algorithm

Theorem

Prim's algorithm produces a minimum spanning tree for a connected graph G.

Proof of Kruskal's algorithm

Theorem

Kruskal's algorithm produces a minimum spanning tree for a connected graph G.

Outline

Implementation of Prim's algorithm

Exercise

Give an implementation of Prim's algorithm (remember Dijkstra's algorithm).

Complexity of implementation

Outline

Implementation of Kruskal's algorithm

Main problem

When chosing an edge $e = (v, w)$ to be possibly added, how to efficiently know if there is already a path between v and w?

Complexity of the implementation

Outline

The problem

The TSP is a simple problem: given a set of cities V and the distances between each pair of cities, what is the shortest possible path that visits each city and returns to the first one?

A complex problem?

A brute-force solution to the TSP consists in enumerating all possible permutations of V.

The associated complexity is of course $O(|V|!)$.

Optimistic hypothesis: 10^{-12} s to compute and evaluate a path.

# of towns	Computation time (s)
10	$3.63 \cdot 10^{-6}$
15	1.31
20	2432902
30	265252859812191058636

Some intuitions...

- 2432902 s $\approx$ 0.77 year
- 265252859812191058636 s ≈ 84111130077 millenia $=$ more than **500 times the age of the Universe**!

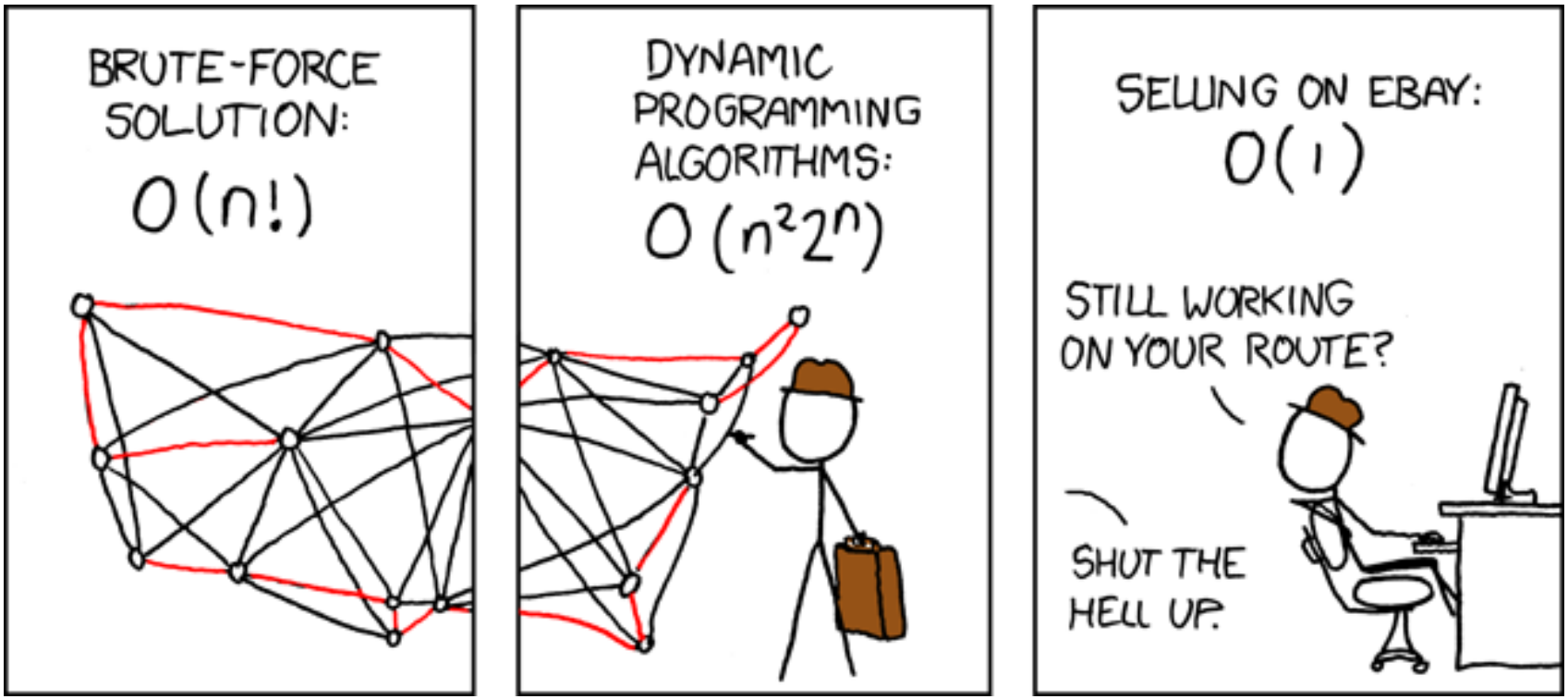
A complex problem?

Some complexity results:

- the TSP problem is NP-hard
- the associated decision problem, i.e. "given a TSP problem and a number x, is there a round-trip route cheaper that x?", is NP-complete

Therefore, we should expect some really hard instances to solve...

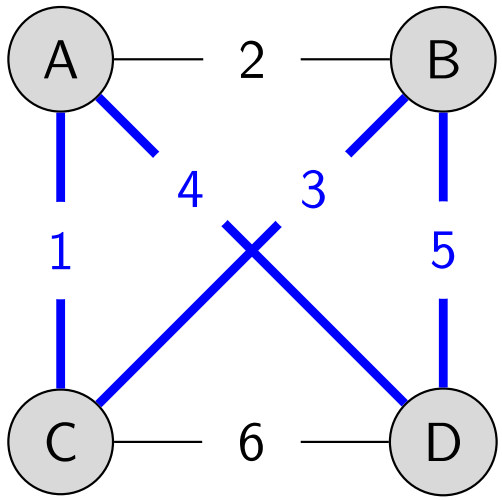
A complex problem?



Representation as a graph

Of course, the TSP can be represented by a graph $G = \langle V, E \rangle$ such that:

- G is not directed
- G is complete (can be relaxed)
- edge costs on edges of E are positive



A dynamic programming solution

Brute-force algorithm takes $O(|V|!)$ time. We will use a dynamic programming algorithm which takes $O(|V|^2 \times 2^{|V|})$ (which is better ☺).

Remember, for a dynamic programming algorithm, you need two properties:

- **optimal substructure**: the solution of the problem can be built from the optimal solutions of its subproblems
- **overlapping subproblems**: the solutions of the subproblems can be reused several times

Outline

Why shortest paths again?

The TSP problem seems to have some relation to the shortest path problem, but with more constraints.

Let us rediscover shortest paths with a **dynamic programming** point of view.

We will look at Bellman-Ford's algorithm, an algorithm that allows **negative costs** on edges.

Shortest paths with negative edges

Problem: if a path $u \to v$ contains a negative cycle, then there is no shortest path from u to v.

So, what can we do if a graph has a negative cycle?

1. compute the shortest path from u to v with cycles allowed
2. compute shortest cycle-free path from u to v
3. assume input graph has no negative cycles

What do you think of the previous proposals?

A dynamic programming solution for shortest paths?

Remember that the main idea for DP is to find an optimal substructure relation between smaller subproblems and larger ones.

Here, subpaths of a shortest path should be shortest paths ☺

Question: how to define "smaller" and "larger" subproblems?

➥ we will use the number of permitted edges in a path

Optimal substructure

Lemma

For every $v \in V$, for every $i \in \mathbb{N}^*$, let P_i be the shortest path $u \to v$ with at most i edges. Then

1. *if P_i has less than $i-1$ edges, then it is a path with at most $i-1$ edges*
2. *if P_i has i edges with a last edge (w, v), then there is a shortest path $P'_{i-1} = P_i - \{(w, v)\}$ $u \to w$ with a most $i-1$ edges*

Optimal substructure

Lemma

For every $v \in V$, for every $i \in \mathbb{N}^*$, let P_i be the shortest path $u \to v$ with at most i edges. Then

1. *if P_i has less than $i-1$ edges, then it is a path with at most $i-1$ edges*
2. *if P_i has i edges with a last edge (w,v), then there is a shortest path $P'_{i-1} = P_i - \{(w,v)\}$ $u \to w$ with a most $i-1$ edges*

How many candidates are there for an optimal solution to a subproblem to destination v?

2	☐
$1+\mathit{indegree}(v)$	☐
$\lvert V\rvert - 1$	☐
$\lvert V\rvert$	☐

Recurrence relation

Let $L_{i,v}$ be the minimum length of a $u \to v$ path with less than i edges with cycles allowed and $+\infty$ if no such path exists.

For every $v \in V$ and every $i \in \mathbb{N}^*$:

$$L_{i,v} = min\{L_{i-1,v}, min_{(w,v)\in E}\{L_{i-1,w} + c_{(w,v)}\}\}$$

- $L_{i-1,v}$: first case (there is a shorter path to v)
- $min_{(w,v)\in E}\{L_{i-1,w} + c_{(w,v)}\}$: case 2

Correctness: brute-force search on the $1 + indegree(v)$ candidates.

A bound on the number of edges

Let us consider a directed graph $G = \langle V, E \rangle$ **with no negative cycles** and $u \in V$. Which is the strongest true statement among the following?

- for every $v \in V$, there is a shortest path $u \to v$ with at most $|V| - 1$ edges ☐
- for every $v \in V$, there is a shortest path $u \to v$ with at most $|V|$ edges ☐
- for every $v \in V$, there is a shortest path $u \to v$ with at most $|E|$ edges ☐
- shortest paths can have an arbitrary large number of edges ☐

Implementation of Bellman-Ford algorithm

A direct implementation of the recurrence relation can be done with an array `A[i, v]`.
This implementation as a running time complexity of $O(|V| \times |E|)$ and a space complexity of $O(|V|^2)$.

In order to improve space complexity, we can only store `dist[v]` which is the current shortest distance from the origin to `v`, i being only a counter.

Bellman-Ford algorithm: optimized version

Procedure SPBF(G, S): Bellman-Ford shortest path algorithm

Input: a graph G with edges labelled with numbers and a vertex S

Result: at the end of the algorithm, the nodes reachable from S are known as well as their predecessor on the shortest path from S and the length of such shortest path. An error occurs if there is a negative edge cycle.

```
1  foreach vertex v of G do
2  |  dist[v] = ∞ ;
3  |  pred[v] = null_node ;
4  end
5  dist[S] = 0;
6  for i ∈ {1, ..., |V| − 1} do
7  |  foreach edge (u,v) with weight w do
8  |  |  if dist[u] + w < dist[v] then
9  |  |  |  dist[v] = dist[u] + w;
10 |  |  |  pred[v] = u
11 |  |  end
12 |  end
13 end
14 foreach edge (u,v) with weight w do
15 |  if dist[u] + w < dist[v] then
16 |  |  error: negative edge cycle!
17 |  end
18 end
```

Outline

A first decomposition into subproblems

A first idea: copy the Bellman-Ford algorithm!

For every $v \in V$ and every $i \in \{1, \ldots, |V|\}$, let $L_{i,v}$ be the minimum length of a $u \to v$ path with less than i edges.

What prevents using these subproblems to obtain a polynomial-time algorithm for TSP?

- there is a superpolynomial number of subproblems ☐
- can't efficiently compute solutions to bigger subproblems from smaller ones ☐
- solving all subproblems does not solve TSP ☐
- nothing ☐

A second decomposition...

OK, let us correct the number of egdes...

For every $v \in V$ and every $i \in \{1, \ldots, |V|\}$, let $L_{i,v}$ be the minimum length of a $u \to v$ path with **exactly** i edges.

What prevents using these subproblems to obtain a polynomial-time algorithm for TSP?

- there is a superpolynomial number of subproblems ☐
- can't efficiently compute solutions to bigger subproblems from smaller ones ☐
- solving all subproblems does not solve TSP ☐
- nothing ☐

A third decomposition...

OK, let us correct again the proposition...

For every $v \in V$ and every $i \in \{1, \ldots, |V|\}$, let $L_{i,v}$ be the minimum length of a $u \to v$ path with **exactly** i edges and **no repeated vertices**.

What prevents using these subproblems to obtain a polynomial-time algorithm for TSP?

- there is a superpolynomial number of subproblems ☐
- can't efficiently compute solutions to bigger subproblems from smaller ones ☐
- solving all subproblems does not solve TSP ☐
- nothing ☐

An optimal substructure

In the following, we suppose for readibility that vertices are named $\{1, \ldots, n\}$. We also suppose that we start the tour with vertex 1.

For every $j \in \{2, \ldots, n\}$, every **subset** $S \subseteq \{2, \ldots, n\}$ that contains j, let $L_{S,j}$ be the minimum length of a path $1 \to j$ visiting exactly once the vertices in S.

Optimal substructure: let P be a shortest path from 1 to j that visits the vertices in S exactly once. If the last edge in P is (k, j), then $P' = P - \{(k, j)\}$ is a shortest path from 1 to k that visits the vertices in $S - \{j\}$ exactly once.

Hence the recurrence:

$$L_{S,j} = \min_{k \in S, k \neq j} \{L_{S-\{j\},k} + c_{(k,j)}\}$$

The Held-Karp algorithm

Function HK(G): Held-Karp algorithm for TSP

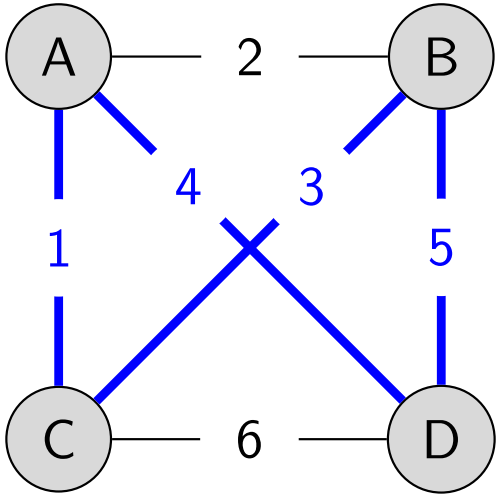
Input: a complete graph G with edges labelled with positive numbers representing distances between vertices

Output: the length of the shortest tour for G

```
1  L is a 2D array indexed by subsets S without 1 and destination j;
2  for i ∈ {2, ..., n} do
3  |  L[{i}, i] = c_(1,i) ;
4  end
5  for m ∈ {2, ..., n − 1} do
6  |  foreach set S ⊆ {2, ..., n} of size m do
7  |  |  foreach j in S do
8  |  |  |  L[S, j] = min_{k∈S,k≠j} {L[S- {j }, k] + c_(k,j)} ;
9  |  |  end
10 |  end
11 end
12 return min_{j∈{2,...,n}} {L[{2, ...n}, j] + c_(j,1)} ;
```

Apply it!

Apply the algorithm on the following graph:



Implementation of Held-Karp algorithm

The main difficulty when implementing Held-Karp algorithm is to manage used space, because the space complexity of the algorithm is $O(2^{|V|})$.

For instance, let us suppose that you have 32 towns, that you store an 4 bytes integer representing the distance in the array, then you need 16GB of memory...

Some remarks:

- when computing path for subsets of size s, you only need the paths for subsets of size $s-1$
  - ➥ not really needed to store all paths values
- subsets can be efficiently encoded as bit vectors (think about it)

You may also need to store the predecessor of a vertex in the tour, but this is not really complicated...

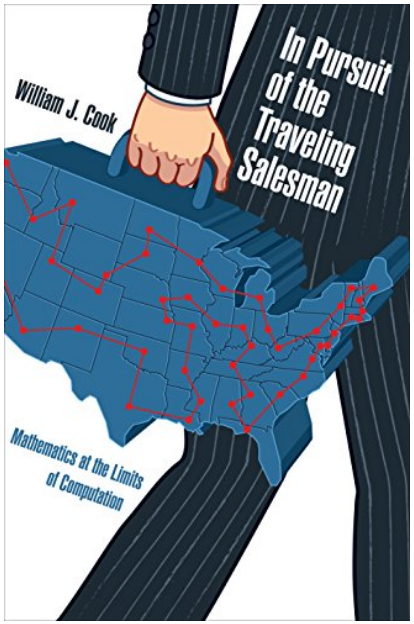
Outline

Solving huge TSP problems

- approximate algorithms exist, one of the most famous being Christofides algorithm that guarantees a solution with a factor $\frac{3}{2}$ of the optimal solution.
  It is based on minimum spanning trees and supposes that edges respect the triangle inequality.
- Concorde is a software written in C and can solve TSP instances with 80,000 towns.
  It is based on linear programming with cutting planes.

Applegate, David et al. (2005).
Concorde.
http://www.math.uwaterloo.ca/tsp/concorde/index.html.

History



Cook, William J. (2011).
In pursuit if the traveling salesman: mathematics at the limits of computation.
Princeton University Press.

http://www.math.uwaterloo.ca/tsp/us/history.html

Credits

- image of the "Car 54" contest: http://www.math.uwaterloo.ca/tsp/us/img/car54.jpg, courtesy of Procter & Gamble
- XKCD's Travelling Salesman Problem: https://www.xkcd.com/399/, Creative Commons Attribution-NonCommercial 2.5 License

Outline

On graph theory (more maths than CS)


Trudeau, Richard J. (1993).
Introduction to graph theory.
Dover Publications.


West, Douglas B. (2001).
Introduction to graph theory.
Second edition.
Pearson Education.


Papadimitriou, Christos H. and Kenneth Steiglitz (1998).
Combinatorial optimization: algorithms and complexity.
Dover Publications Inc.

On algorithms on graphs (CS with maths)


Cormen, Thomas H. et al. (2009).
Introduction to algorithms.
Third edition.
MIT Press.


Kleinberg, Jon and Éva Tardos (2006).
Algorithm design.
Pearson Education.


Sedgewick, Robert and Kevin Wayne (2011).
Algorithms.
Fourth edition.
Addison-Wesley Professional.
https://algs4.cs.princeton.edu/home/.