Dynamic programming

Johan Montelius

KTH

VT23

Hinges and latches

Assume you're producing hinges and latches and would like to make as much money as possible.

- Your resources are 2400g of raw material and 480 minutes of time.
- Each hinge takes 260g of material and 40 minutes to make.
- Each latch takes 180g of material and 60 minutes to make.
- Hinges are sold for 30 crowns and latches for 24 crowns.

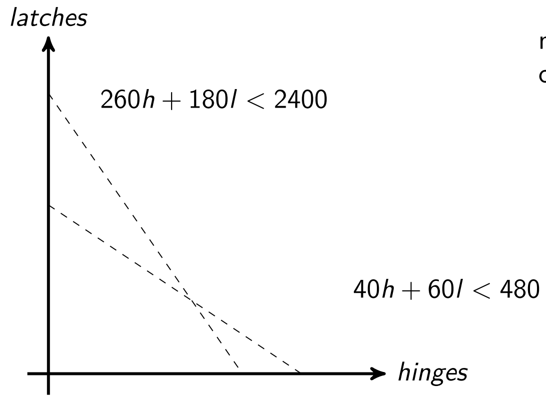
Hinges and latches

Assume you're producing hinges and latches and would like to make as much money as possible.

Assume we make h hinges and l latches:

- limited resources: $260h + 180l < 2400$
- limited time: $40h + 60l < 480$
- profit: $p = 30h + 24l$
- find h and l to maximize p

linear programming



$$p = 30h + 24l$$

maxium profit is found in one of the corners:

$$h = 0, l = 8 \rightarrow p = 192$$

$$h = 9, l = 0 \rightarrow p = 270$$

$$h = 7, l = 3 \rightarrow p = 282$$

search for the answer

To find the maximum profit, we either:

- make a hinge and then maximize profit or
- make a latch and then maximize profit.

search for the answer

Describe a product as `{material, time, prize}`: a hinge is `{260, 40, 30}` and a latch is `{180, 60, 24}`.

Define a function `search(material, time, hinge, latch)`, that given an amount of material, time and descriptions of hinges and latches, returns the number of hinges, *h*, and latches, *l*, to produce to maximize profit *p*, `{h, l, p}`.

```
@spec seach(integer, integer, hinge, latch) :: {integer, integer, integer}

def search(material, time, hinge, latch) do
   :
   :
  {hinges, latches, profit}
end
```


search for the answer

```
def search(m, t, {hm, ht, hp}=h, {lm, lt, lp}=l) when (m >= hm) and
                                                     (t >= ht) and
                                                     (m >= lm) and
                                                     (t >= lt) do

  ## we have material and time to make either a hinge or latch
  {hi, li, pi} = search((m-hm), (t-ht), h, l)
  {hj, lj, pj} = search((m-lm), (t-lt), h, l)

  ## which alternative will give us the maximum profit
  if (pi+hp) > (pj+lp) do
    ## make hinge
    {(hi+1), li, (pi+hp)}
  else
    # make a latch
    {hj, (lj+1), (pj+lp)}
  end
end
```


search for the answer

```
   :
def search(m, t, {hm, ht, hp}=h, l) when (m >= hm) and (t >= ht)  do
  ## we can make a hinge
  {hn, ln, p} = search((m-hm), (t-ht), h, l)
  {hn+1, ln, (p+hp)}
end

def search(m, t, h, {lm, lt, lp}=l) when (m >= lm) and (t >= lt)  do
  ## we can make a latch
  {hn, ln, p} = search((m-lm), (t-lt), h, l)
  {hn, ln+1, p+lp}
end

def search(_, _, _, _) do
  ## we can make neither
  {0,0,0}
end
```

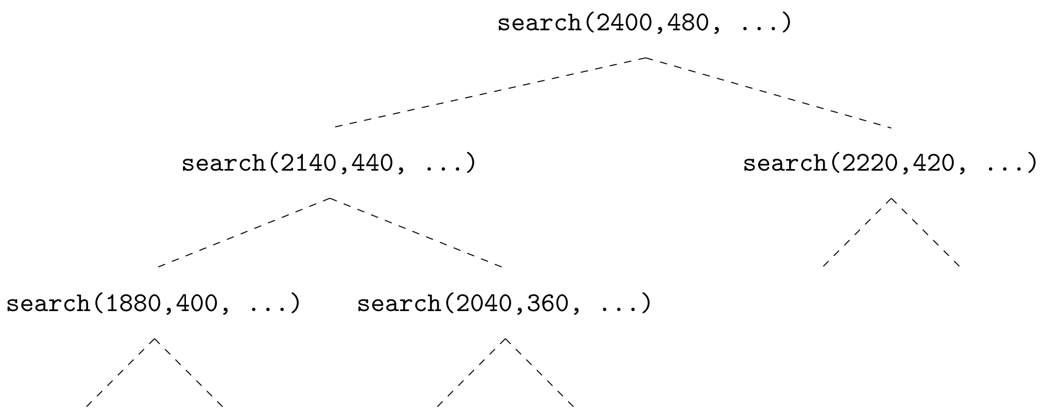
a test

```
>Hinges.search(2400, 480, {260, 40, 30}, {180, 60, 24})
{7, 3, 282}
>Hinges.search(2000,480,{260,40,30},{180,60,24})
{4,5,240}
>Hinges.search(2800,520,{260,40,30},{180,60,32})
{7,4,338}
```


problem solved

What is the problem?

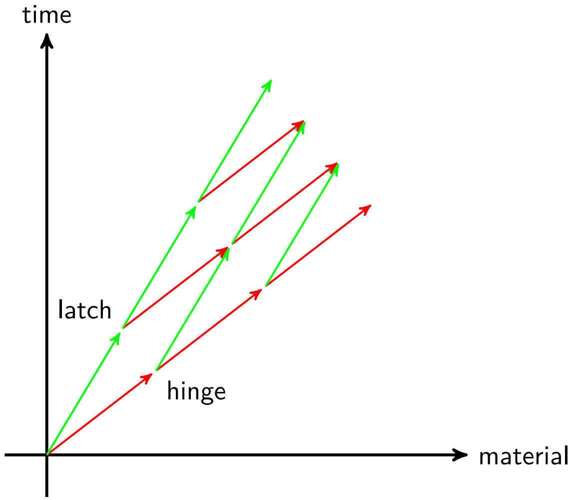
complexity



What is the depth of this tree? How does it relate to the size of the resources?

complexity

the m x t space



dynamic programming

Problem divided into simpler parts that can be solved independently, but
- the parts share sub-problems that can be reused.

Fibonacci

```
def fib(0) do 0 end
def fib(1) do 1 end
def fib(n) do
    fib(n-1) + fib(n-2)
end
```

```
def fib(0) do {0, nil} end

def fib(1) do {1, 0} end

def fib(n) do
    {n1, n2} = fib(n-1)

    {n1+n2, n1}
end
```


memory

Let's add a memory to the search function.

```
def memory(material, time, hinge, latch) do
  mem = Memory.new()
  {solution, _} = search(material, time, hinge, latch, mem)
  solution
end
```

```
def check(material, time, hinge, latch, mem) do
  case Memory.lookup({material,time}, mem) do
    nil ->
      ## no previous solution found
      {solution, mem} = search(material, time, hinge, latch, mem)
      {solution, Memory.store({material,time}, solution, mem)}
    found ->
      {found, mem}
  end
end
```

memory

```
def search(m, t,..., mem) when ... do

    {..., mem} = check(...,  mem)

    {..., mem} = check(..., mem)

    if ...  do
      {..., mem}

    else
      {..., mem}
    end
  end
```


a memory

the *key* is a tuple `{m,t}`, defining the remaining resource (the point in the mxt space).

The *value* is the number of hinges and latches and best profit possible at this point `{h, l, p}`.

The functions we should implement are:

- `new()`: returns a new memory
- `store(k, v, mem)`: returns a new memory where the key `k` is associated with the value `v`
- `lookup(k, mem)`: return the value `v` assocaued with the key or `nil` if not found

a key-value list

Let's implement the memory as a list of tuples `{k, v}`.

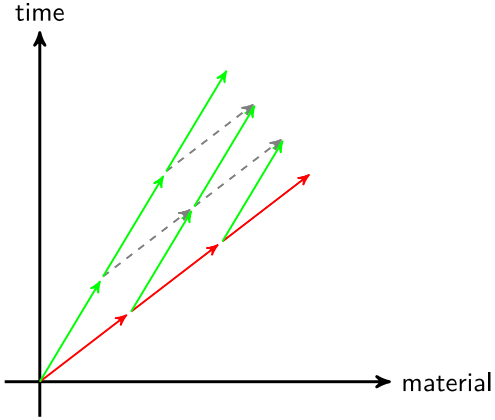
```
defmodule Memory do

def new() do [] end


def store(k, v, mem) do

   [{k, v}|mem]
end


def lookup(_, []) do nil end

def lookup(k, [{k,v}|_]) do v end

def lookup(k, [_|rest]) do lookup(k, rest) end
```


benchmark

on a i7-4500 1.8GHz, time in ms

m	t	m + t	search	memory
1000	200	1200	0.01	0.03
2000	400	2400	0.08	0.08
3000	600	3600	0.70	0.13
4000	800	4800	10	0.35
5000	1000	6000	110	0.42
6000	1200	7200	1900	0.80
7000	1400	8400	32000	1.30
8000	1600	9600	550000	2.10

complexity




```
def lookup(_, []) do nil end
def lookup(k, [{k,v}|_]) do v end
def lookup(k, [_|rest]) do
  lookup(k, rest)
end
```


a map

Why not implement the memory as a hash map?

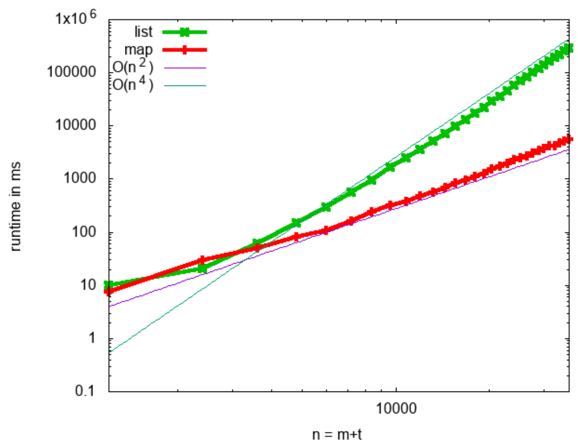
```
defmodule Better do

def new() do %{} end

def store(k,v, mem) do
  Map.put(mem, k, v)
end

def lookup(k, mem) do
  Map.get(mem, k)
end
```


benchmark

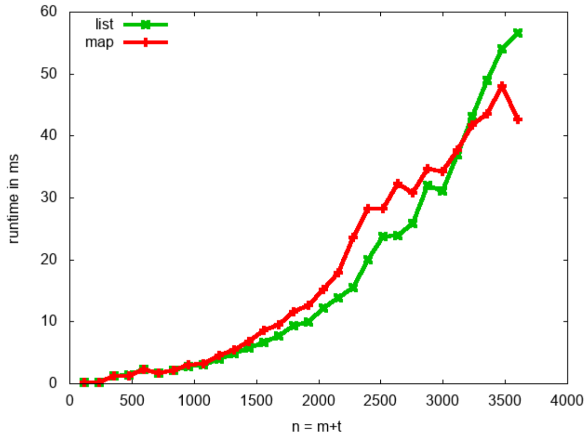
on a i7-4500 1.8GHz, time in ms

m	t	m+t	list	map
1000	200	1200	0.03	0.06
2000	400	2400	0.11	0.18
3000	600	3600	0.34	0.24
4000	800	4800	0.82	0.39
5000	1000	6000	1.26	0.31
6000	1200	7200	1.53	0.36
7000	1400	8400	2.25	0.34
8000	1600	9600	2.94	0.43
9000	1800	10800	4.22	0.49
10000	2000	12000	6.22	0.58
11000	2200	13200	8.97	0.69
12000	2400	14400	12.55	0.84

benchmark



same benchmark



dynamic programming

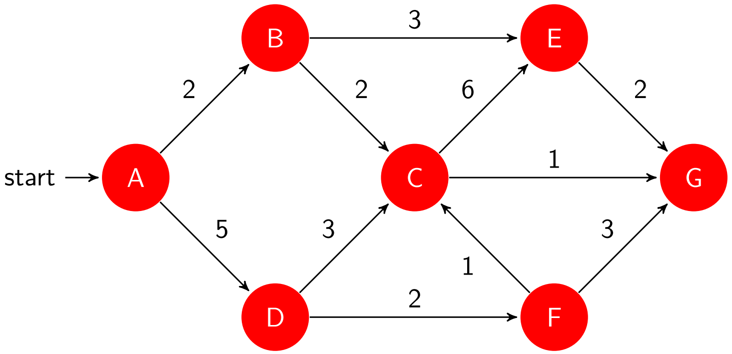
Problem divided into simpler parts that can be solved independently, but

- the parts share subproblems that can be reused and,

- we can memorize solutions of subproblems.

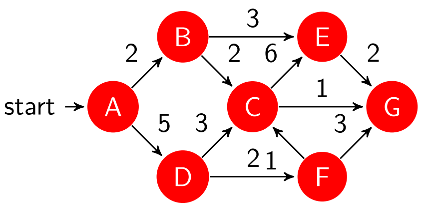
shortest path

Find the shortest path from one node to another.



We assume the graph is a "Directed Acyclic Graph" (DAG)

dynamic programming aproach



The *dynamic programming approach*:

- find a recursive solution
- memorize solutions to subproblems

dynamic programming aproach

If we are in the final node the distance is zero and the path is

```
def shortest(from, from, _) do  {0, []} end
```

.

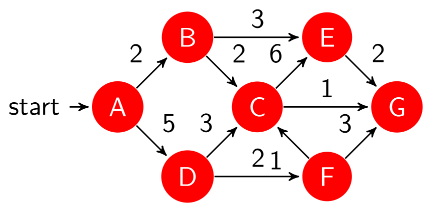
Otherwise, for each outgoing edge: find the shortest path from the reached node and return the shortest given the distance to the node.

```
def shortest(from, to, graph) do
  next = Graph.next(from, graph)
  distances = distances(next, to, graph)
  select(distances)
end
```

If no path is found we should return `{:inf, nil}`.

a graph

How do we represent a graph?



As a list of edges:

```
[{:a, :b, 2}, {:a, :d, 5}, {:b, :c, 2}
...  ]
```

As a list of nodes:

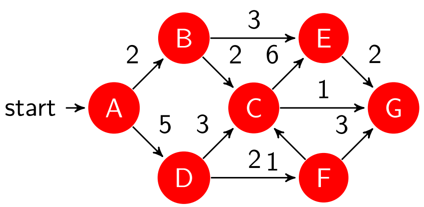
```
[{:a, [{:b, 2}, {:d, 5}]},
 {:b, [{:c, 2}, {:e, 3}]},
 ...]
```

As a matrix of edges:

```
{{ nil, 2 ,nil, 5 ,nil,nil,nil},
 { nil,nil, 2 ,nil, 3 ,nil,nil},
  ...}}
```


a graph

How about this?




```
g = {:g, []}
e = {:e, [{g , 2}]}
c = {:c, [{g, 1}, {e, 6}]}
f = {:f, [{c, 1}, {g, 3}]}
d = {:d, [{f, 2}, {c, 3}]}
b = {:b, [{c, 2}, {e, 3}]}
a = {:a, [{b, 2}, {d, 5}]}

[a: a, b: b, c: c, d: d, e: e, f: f, g: g]
```

What has this to do with topological order?

the graph

Assume we represent a graph by a map indexd by nodes. Each node holds a key-value list of edges.

```
defmodule Graph do

  def sample() do
    new([a: [b: 2, d: 5], b: [c: 2], ... ])
  end

  def new(nodes) do
    Map.new(nodes)
  end

  def next(from, map) do
    Map.get(map, from, [])
  end
end
```

distances

Find the distance to the destination from each of the next steps.

```
def distances(next, to, graph) do

    Enum.map(next, fn({n,d}) ->

                   case shortest(n, to, graph) do

                        {:inf, nil} -> {:inf, nil}

                        {k, path} -> {d+k, [n|path]}

                   end
                end)
end
```


select

Select the smallest path in the list: `[{9, [:d, :c, :g]}, ..]`

```
def select(distances) do
  List.foldl(distances,
    {:inf, nil},
    fn ({d,_}=s,{ad,_}=acc) ->
      if d < ad do
        s
      else
        acc
      end
    end)
end
```

If the list is empty, the result could be `{:inf, nil}`.

dynamic programming aproach

If we are in the final node, the distance is zero and the path is

```
def shortest(from, from, _) do {0, []} end
```

.

Otherwise, for each outgoing edge: find the shortest path from the reached node and return the shortest given the distance to the node.

```
def shortest(from, to, graph) do
  next = Graph.next(from, graph)
  distances = distances(next, to, graph)
  select(distances)
end
```

What is the complexity?

let's add a memory

```
def dynamic(from, to, graph) do
  mem = Memory.new()
  {solution, _} = shortest(from, to, graph, mem)
  solution
end
```

shortest(from, to, graph, mem) should return {shortest path, updated memmory}

```
def shortest(from, from, _, mem) do
    {{0, []}, ...}
end
def shortest(from, to, graph, mem) do
    next = Graph.next(from, graph)
    {..., ...} = distances(next, to, graph, mem)
    shortest = select(...)
    {..., ...}
end
```

shortest path given memeory

For all next steps, find the shortesta path.

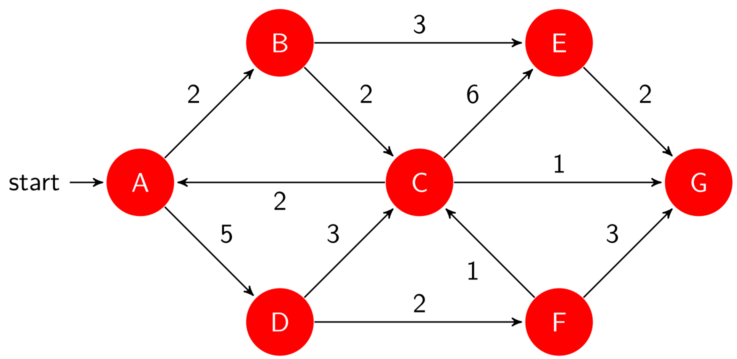
```
def distances(next, to, graph, mem) do
  List.foldl(next, {[], mem},
    fn ({t,d}, {dis,mem}=acc) ->
      case check(t, to, graph, mem) do
        {{:inf, _}, _} ->
          acc
        {{n, path}, mem} ->
          {[{d+n, [t|path]}| dis ], mem}
      end
    end)
end
```


shortest path given memeory

If a solution exists use it, if not - compute it.

```
def check(from, to, graph, mem) do
  case Memory.lookup(from, mem) do
    nil ->
      {solution, mem} = shortest(from, to, graph, mem)
      {solution,  Memory.store(from, solution, mem)}
   solution ->
      {solution, mem}
  end
end
```


what if?



Summary

Problem divided into simpler parts that can be solved independently, but

- the parts share subproblems that can be reused and,

- we can memorize solutions of subproblems.