Lecture 14

Algorithms & Data Structures

Goldsmiths Computing

February 4, 2019

Outline

Outline

Lecture

1. Graphs
2. Spanning trees
3. Path finding
    - (or at least half of it)

Lab

Heaps!

- two different constructors (`incremental` boolean)
- heapsort

VLE activities

Implicit data structures quiz

Statistics so far:

- A attempts: average mark B
- C students: average mark D
    - E under 4.00, F over 6.99, G at 10.00

Quiz closes at 16:00 on Friday 2nd February

- no extensions
- grade is
    - 0 (for no attempt)
    - $30 + 70 \times (\text{score}/10)^2$

VLE activities (cont'd)

Binary search quiz

VLE activities (cont'd)

Binary search quiz

VLE activities (cont'd)

Binary search quiz

Outline

Motivation

- Exploration of known, partially known or unknown surroundings
- Component of various AI solutions
  - especially agents exploring some space:
    - ... game enemies
    - ... NPCs
    - ... self-driving cars

Definition

Single-source shortest path

- from a single source node:
  - find the shortest path to every node in the graph
  - stop early if we have a specific target node

Basic approach

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."

Basic approach

> *"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."*

Initial state

Start at the start node

Basic approach

> *“Begin at the beginning,” the King said gravely, “and go on till you come to the end: then stop.”*

Initial state

Start at the start node

Goal state

Stop when we have found a path (optimally: shortest) to the target node

- (if no target: stop when there are no nodes without the shortest path known)

Basic approach

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."

Initial state

Start at the start node

Goal state

Stop when we have found a path (optimally: shortest) to the target node

- (if no target: stop when there are no nodes without the shortest path known)

State expansion

Explore the graph using neighbours of already-visited nodes

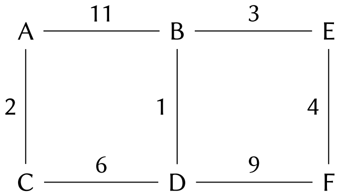
Greedy best-first search

Explore the graph using the neighbour of the current node which is closest to the target.

function GBFS(G,start,end)
  current $\leftarrow$ start
  result $\leftarrow$ **new** queue()
  while current $\neq$ end **do**
    ENQUEUE(result,current)
    ns $\leftarrow$ NEIGHBOURS(G,current)
    current $\leftarrow \arg\min_{n \in \text{ns}} d(n, \text{end})$
  end while
  return result
end function

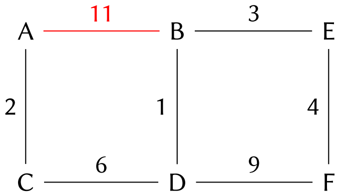
Example

Node	d(n,F)
A	14
B	6
C	12
D	7
E	4
F	0



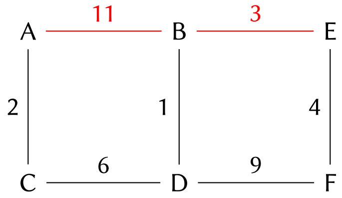
Example

Node	d(n,F)
A	14
B	6
C	12
D	7
E	4
F	0



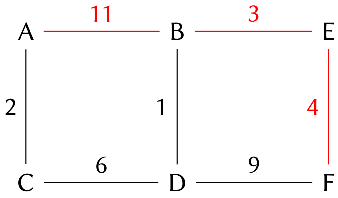
Example

Node	$d(\text{n,F})$
A	14
B	6
C	12
D	7
E	4
F	0



Example

Node	d(n,F)
A	14
B	6
C	12
D	7
E	4
F	0



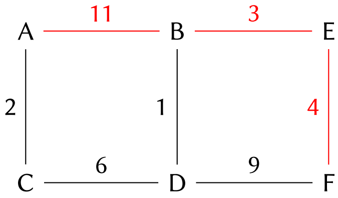
Result

path A ⟶ B ⟶ E ⟶ F

distance 18

Example

Node	d(n,F)
A	14
B	6
C	12
D	7
E	4
F	0



Result

path A ⟶ B ⟶ E ⟶ F

distance 18

Problems

- does not necessarily find a solution!
- not guaranteed optimal

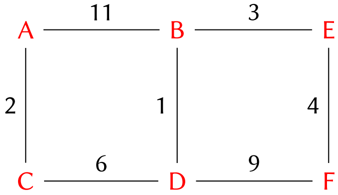
Dijkstra's algorithm

Explore the graph using the neighbour of the already-visited nodes with the smallest distance from the start node

```
function DIJKSTRA(G,start,end)
    dist ← new table(); prev ← new table()
    Q ← new min-heap(dist)
    for v ∈ G do
        dist[v] ← 0 if v = start else ∞; INSERT(Q,v)
    end for
    while ¬ EMPTY(Q) do
        u ← EXTRACT-MIN(Q)
        if u = end then
            s ← new stack()
            while u ≠ start do
                PUSH(s,u); u ← prev[u]
            end while
            return s
        end if
        for v ∈ NEIGHBOURS(G,u) do
            d ← dist[u] + WEIGHT(G,u,v)
            if d < dist[v] then
                dist[v] ← d; prev[v] ← u; DECREASE-KEY(Q,v,dist[v])
            end if
        end for
    end while
end function
```

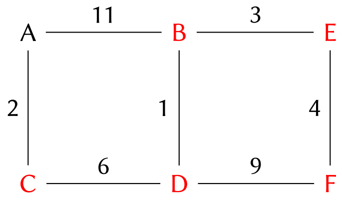
Example

Node	dist[n]	prev[n]
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	



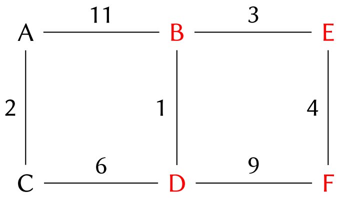
Example

Node	dist[n]	prev[n]
A	0	
B	11	A
C	2	A
D	∞	
E	∞	
F	∞	



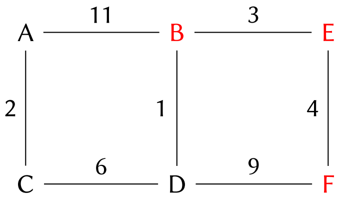
Example

Node	dist[n]	prev[n]
A	0	
B	11	A
C	2	A
D	8	C
E	∞	
F	∞	



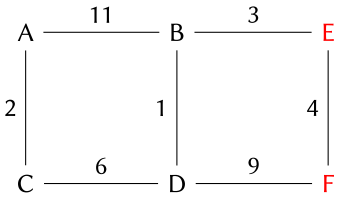
Example

Node	dist[n]	prev[n]
A	0	
B	9	D
C	2	A
D	8	C
E	∞	
F	17	D



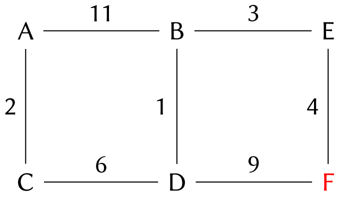
Example

Node	dist[n]	prev[n]
A	0	
B	9	D
C	2	A
D	8	C
E	12	B
F	17	D



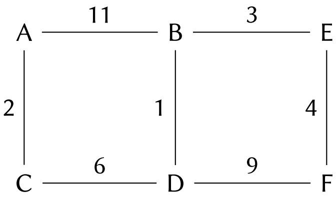
Example

Node	dist[n]	prev[n]
A	0	
B	9	D
C	2	A
D	8	C
E	12	B
F	16	E



Example

Node	dist[n]	prev[n]
A	0	
B	9	D
C	2	A
D	8	C
E	12	B
F	16	E



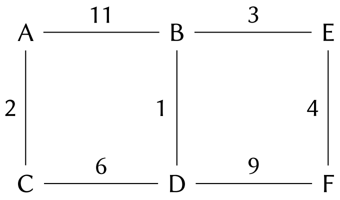
Result

path $A \longrightarrow C \longrightarrow D \longrightarrow B \longrightarrow E \longrightarrow F$

distance 16

Example

Node	dist[n]	prev[n]
A	0	
B	9	D
C	2	A
D	8	C
E	12	B
F	16	E



Result

path $A \longrightarrow C \longrightarrow D \longrightarrow B \longrightarrow E \longrightarrow F$

distance 16

Note

- requires all non-negative weights
- guaranteed to find shortest path
- need priority queue (min-heap) for efficient operation
- does not use distance estimate information

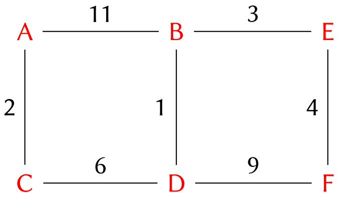
A*

Explore the graph using the neighbour of the already-visited nodes with the smallest estimated distance from the start node to the target node

```
function A*(G,start,end)
    dist ← new table(); prev ← new table()
    Q ← new min-heap(dist)
    for v ∈ G do
        dist[v] ← 0 if v = start else ∞; INSERT(Q,v)
    end for
    while ¬ EMPTY(Q) do
        u ← EXTRACT-MIN(Q)
        if u = end then
            s ← new stack()
            while u ≠ start do
                PUSH(s,u); u ← prev[u]
            end while
            return s
        end if
        for v ∈ NEIGHBOURS(G,u) do
            d ← dist[u] + WEIGHT(G,u,v) + H(v)
            if d < dist[v] then
                dist[v] ← d; prev[v] ← u; DECREASE-KEY(Q,v,dist[v])
            end if
        end for
    end while
end function
```

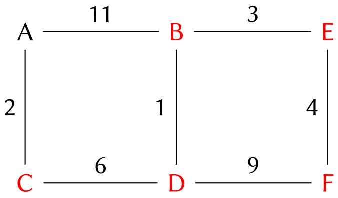
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	∞	
C	12	∞	
D	7	∞	
E	4	∞	
F	0	∞	



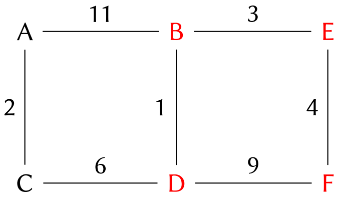
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	11	A
C	12	2	A
D	7	∞	
E	4	∞	
F	0	∞	



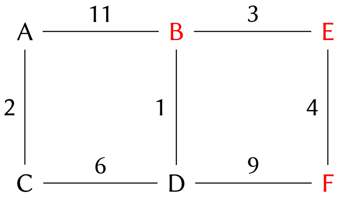
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	11	A
C	12	2	A
D	7	8	C
E	4	∞	
F	0	∞	



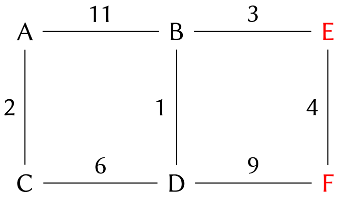
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	9	D
C	12	2	A
D	7	8	C
E	4	∞	
F	0	17	D



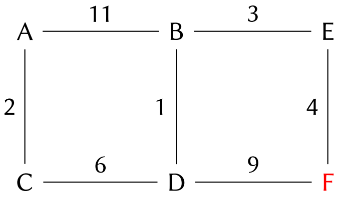
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	9	D
C	12	2	A
D	7	8	C
E	4	12	B
F	0	17	D



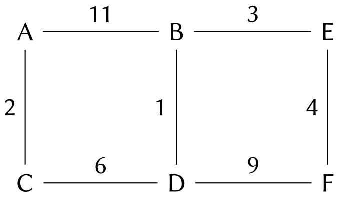
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	9	D
C	12	2	A
D	7	8	C
E	4	12	B
F	0	16	E



Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	9	D
C	12	2	A
D	7	8	C
E	4	12	B
F	0	16	E



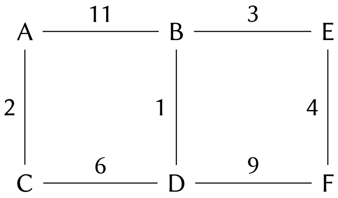
Result

path A → C → D → B → E → F

distance 16

Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	9	D
C	12	2	A
D	7	8	C
E	4	12	B
F	0	16	E



Result

path A ⟶ C ⟶ D ⟶ B ⟶ E ⟶ F

distance 16

Note

- generalisation of Dijkstra’s algorithm
- distance estimation H must be admissible
  - lower bound
  - non-negative
  - (Dijkstra’s algorithm is A* with H(n) = 0)

Work

1. Reading
   - CLRS, chapter 24
   - Drozdek, sections 8.2, 8.3
2. Questions from CLRS

   Exercises 24.3-1

Outline

Introduction

Path finding

Memoization

Dynamic programming

Motivation

We've seen a trade-off between space and time in various places so far. Is there a systematic way of thinking about it?

Definition

Memoization is the use of some data structure to store the results of previous computations, particularly when those results will be re-used. (Similar: cacheing)

Example: factorial

$$n! = \begin{cases} 1 & n < 2 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

```
function FACT(n)
    if n < 2 then
        return 1
    else
        return n × FACT(n−1)
    end if
end function
```

Complexity

Example: factorial

$$n! = \begin{cases} 1 & n < 2 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

```
function FACT(n)
    if n < 2 then
        return 1
    else
        return n × FACT(n−1)
    end if
end function
```

Complexity

time $\Omega(N)$

space $\Omega(N)$

Example: factorial (accumulator)

save stack space: use accumulator instead

```
function FACT(n)
    return FACTAUX(n,1)
end function
function FACTAUX(n,r)
    if n < 2 then
        return r
    else
        return FACTAUX(n−1,n×r)
    end if
end function
```

Complexity

Example: factorial (accumulator)

save stack space: use accumulator instead

```
function FACT(n)
    return FACTAUX(n,1)
end function
function FACTAUX(n,r)
    if n < 2 then
        return r
    else
        return FACTAUX(n−1,n×r)
    end if
end function
```

Complexity

time $\Omega(N)$

space $\Omega(1)$

Example: factorial (memoized)

```
T ← new Vector(1000)
for 0 ≤ i < 1000 do
    T ← 0
end for
function FACTMEMO(n)
    if T[n] > 0 then
        return T[n]
    else if n < 2 then
        T[n] ← n; return T[n]
    else
        T[n] ← n × FACTMEMO(n−1); return T[n]
    end if
end function
```

Complexity

Example: factorial (memoized)

```
T ← new Vector(1000)
for 0 ≤ i < 1000 do
    T ← 0
end for
function FACTMEMO(n)
    if T[n] > 0 then
        return T[n]
    else if n < 2 then
        T[n] ← n; return T[n]
    else
        T[n] ← n × FACTMEMO(n−1); return T[n]
    end if
end function
```

Complexity

time $\Omega(N)$ (first time); $\Theta(1)$ (subsequent times)

space $\Omega(N)$

Example: Fibonacci

$$u_n = \begin{cases} n & n < 2 \\ u_{n-1} + u_{n-2} & \text{otherwise} \end{cases}$$

```
function Fib(n)
    if n < 2 then
        return n
    else
        return Fib(n−1) + Fib(n−2)
    end if
end function
```

Complexity

time $\Omega(\varphi^N)$

space $\Omega(\varphi^N)$

Example: Fibonacci (memoized)

```
T ← new Vector(1000)
for 0 ≤ i < 1000 do
    T ← −1
end for
function FibMemo(n)
    if T[n] ≥ 0 then
        return T[n]
    else if n < 2 then
        T[n] ← n
        return T[n]
    else
        T[n] ← FibMemo(n−1) + FibMemo(n−2)
        return T[n]
    end if
end function
```

Work

1. Reading
   - CLRS, chapter 15
2. Exercises and Problems

   Exercises from CLRS 15.1-1, 15.1-4

Outline

Motivation

Technique for applying memoization to optimization problems.

- not really “dynamic”;
- not really “programming” (as we understand it today).

Marketing!

Definition

The bottom-up application of memoization (stored computation) to solve problems searching for an optimum (shortest, smallest, ...) of a set of possibilities, where the optimum can be described in terms of subproblems.

Example: factorial

$$n! = \begin{cases} 1 & n < 2 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

```
function FACT(n)
    if n < 2 then
        return 1
    else
        return n × FACT(n−1)
    end if
end function
```

Complexity

Example: factorial

$$n! = \begin{cases} 1 & n < 2 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

```
function FACT(n)
    if n < 2 then
        return 1
    else
        return n × FACT(n−1)
    end if
end function
```

Complexity

time $\Omega(N)$

space $\Omega(N)$

Example: factorial (memoized)

```
T ← new Vector(1000)
for 0 ≤ i < 1000 do
    T ← 0
end for
function FACTMEMO(n)
    if T[n] > 0 then
        return T[n]
    else if n < 2 then
        T[n] ← n; return T[n]
    else
        T[n] ← n × FACTMEMO(n−1); return T[n]
    end if
end function
```

Complexity

Example: factorial (memoized)

```
T ← new Vector(1000)
for 0 ≤ i < 1000 do
    T ← 0
end for
function FACTMEMO(n)
    if T[n] > 0 then
        return T[n]
    else if n < 2 then
        T[n] ← n; return T[n]
    else
        T[n] ← n × FACTMEMO(n−1); return T[n]
    end if
end function
```

Complexity

time $\Omega(N)$ (first time); $\Theta(1)$ (subsequent times)

space $\Omega(N)$

Example: factorial (dynamic programming)

function FACTDP(n)
    T ← new Vector(n+1)
    T[0] ← 1
    for $0 < i \le n$ **do**
        T[i] ← n × T[i-1]
    end for
    return T[n]
end function

Example: Fibonacci

$$u_n = \begin{cases} n & n < 2 \\ u_{n-1} + u_{n-2} & \text{otherwise} \end{cases}$$

function Fib(n)
  if n < 2 **then**
    return n
  else
    return Fib(n−1) + Fib(n−2)
  end if
end function

Complexity

time $\Omega(\varphi^N)$
space $\Omega(\varphi^N)$

Example: Fibonacci (memoized)

```
T ← new Vector(1000)
for 0 ≤ i < 1000 do
    T ← −1
end for
function FibMemo(n)
    if T[n] ≥ 0 then
        return T[n]
    else if n < 2 then
        T[n] ← n
        return T[n]
    else
        T[n] ← FibMemo(n−1) + FibMemo(n−2)
        return T[n]
    end if
end function
```

Example: Fibonacci (dynamic programming)

function FibDP(n)
  T ← new Vector(n+1)
  T[0] ← 0
  T[1] ← 1
  for $1 < i \leq n$ **do**
    T[i] ← T[i-1] + T[i-2]
  end for
  return T[n]
end function

Example: coins

Given a collection of denominations {D}, how many coins does it take to make a particular value v?

- extension: in what way can we make v using the smallest number of coins?

Example: coins

```
function GREEDY(D,v)
    if v = 0 then
        return 0
    else
        cs ← {c|c ∈ D ∧ c ≤ v}
        c ← MAX(cs)
        return 1 + GREEDY(D,v-c)
    end if
end function
```

Example: coins

function $\textsc{Opt}(D,v)$
    if $v \in D$ **then**
        return 1
    else if $v < \textsc{min}(D)$ **then**
        return false
    else
        $cs \longleftarrow \{\textsc{Opt}(D,v\text{-}c) | c \in D \wedge Opt(c) \neq \text{false} \}$
        return $1 + \textsc{min}(cs)$
    end if
end function

Example: coins

```
function LOOKUP(T,i)
    if i < 0 then
        return ∞
    else
        return T[i]
    end if
end function
function OPTDYNAMICPROGRAMMING(D,v)
    T ← new Vector(v)
    T[0] ← 0
    for 0 < i ≤ v do
        cs ← {1 + LOOKUP(T,i-c)|c ∈ D}
        T[i] ← MIN(cs)
    end for
    return T[v]
end function
```

Example: image seam carving

Assume some "energy" measurement for pixels E(i,j)

$$c(i,j) = \begin{cases} E(i,j) & j = 0 \\ E(i,j) + \min(c(i-1,j-1), c(i,j-1), c(i+1,j-1)) & \text{otherwise} \end{cases}$$

```
function SEAM(I)
    w ← WIDTH(I); h ← HEIGHT(I)
    T ← new Array(w+2, h)
    for 0 ≤ i < w do
        T[i+1,j] ← (E(I,i,j),NIL)
    end for
    for 0 ≤ j < h do
        T[0,j] ← (∞,NIL); T[w+1,j] ← (∞,NIL)
    end for
    for 0 < j < h do
        for 0 ≤ i < w do
            T[i+1,j] ← MIN1((T[i,j-1],i), (T[i+1,j-1],i+1), (T[i+2,j-1],i+2))
        end for
    end for
end function
```

Example: edit distance

Operations needed to edit one string into another:

- **insertion** insert a character into the string (cost: ci)
- **deletion** delete a character from the string (cost: cd)
- **substitution** substitute one character for another (cost: cs)

```
function EditDistance(S,Z)
    if length(S) = 0 then
        return ci × length(Z)
    else if length(Z) = 0 then
        return cd × length(S)
    else
        ins ← ci + EditDistance(Z[0]S, Z)
        del ← cd + EditDistance(S[1..], Z)
        if Z[0] = S[0] then
            sub ← EditDistance(S[1..], Z[1..])
        else
            sub ← cs + EditDistance(S[1..], Z[1..])
        end if
        return min(ins, del, sub)
    end if
end function
```

Example: edit distance

Operations needed to edit one string into another:

insertion	insert a character into the string (cost: ci)
deletion	delete a character from the string (cost: cd)
substitution	substitute one character for another (cost: cs)

```
function EditDistance(S,Z)
    if length(S) = 0 then
        return ci × length(Z)
    else if length(Z) = 0 then
        return cd × length(S)
    else
        ins ← ci + EditDistance(S, Z[1..])
        del ← cd + EditDistance(S[1..], Z)
        if Z[0] = S[0] then
            sub ← EditDistance(S[1..], Z[1..])
        else
            sub ← cs + EditDistance(S[1..], Z[1..])
        end if
        return min(ins, del, sub)
    end if
end function
```

Example: edit distance

```
function EditDistanceDP(S,Z)
    ls ← length(S); lz ← length(Z)
    T ← new Array(ls+1, lz+1)
    for 0 ≤ i ≤ ls do
        T[i,0] ← i × cd
    end for
    for 0 ≤ j ≤ lz do
        T[0,j] ← j × ci
    end for
    for 0 < i ≤ ls do
        for 0 < j ≤ lz do
            if S[i-1] = Z[j-1] then
                T[i,j] ← T[i-1,j-1]
            else
                ins ← ci + T[i,j-1]
                del ← cd + T[i-1,j]
                sub ← cs + T[i-1,j-1]
                T[i,j] ← min(ins, del, sub)
            end if
        end for
    end for
    return T[ls,lz]
end function
```

Dynamic programming and memoization

memoization

- small modification of natural recursive definition
- introduction of a cache to store intermediate results
- start from problem, work on progressively smaller cases

dynamic programming

- more substantial rewrite of recursive definition
- introduction of a table to store successive results
- start from base case, work on progressively larger cases

Work

1. Reading
   - CLRS, chapter 15
   - DPV, chapter 6
2. Exercises and Problems

   Exercises from CLRS 15.1-5

   Exercises from DPV 6.1, 6.2

   CLRS 15-4 Printing neatly

   CLRS 15-5 Edit distance