CS 135 — L20 - Directed Graph

Luke Lu • 2025-11-20

🐋 **Info — Directed Graph**

- A directed graph consists of nodes and edges.
- Given an edge v, w, we say w is an **out-neigbour** and v is an **neigbour**.
- A sequence of nodes forms a **path**
- If $v_1 = v_k$, the path is called a cycle
- A directed graph without a cycle is called **directed acyclic graph(DAG)**

Tool for better visualization: https://csacademy.com/app/graph_editor/

Adjacency list representation

We don't use BT for a representation of DAG because there are cases points that are never reachable from one designated "root" node

Example:

```
(define g
 '((A (C D E))
  (B (E J))
  (C ())
  (D (F J))
  (E (K))
  (F (K H))
  (H ())
  (J (H))
  (K ()))
 )
```

Contracts

```
;; A set is (list of Sym)
;; Requires: symbols must be unique

;; A graph is an association list from nodes to their set of
;; out-neigbours

;; A graph is a (listof (list Sym Set))
;; Requires: keys must be unique

;; DAG is a graph
;; Requies graph must be acyclic
```

Finding the out-neigbours

```
;; Look up the out-neighbours of a node
;; neighbours: Sym Graph -> Set
(define (neighbours v g)
 (cond
   [(empty? g) empty]
   [(symbol=? v (first (first g))) (second (first g))]
   [else (neighbours v (rest g))]))
```

Path Finding

> 💡 **Tip — Backtracking**
>
> - Backtracking algorithms try to find a path from an origin to a destination.
> - If the initial attempt does not work, such an algorithm "backtracks" and tries another choice.
> - Eventually, either a path is found, or all possibilities are exhausted, meaning there is no path.

> 🐋 **Info — Path Finding Algo**
>
> These two functions, along with `neigbours` function, construct a path finding algorithm in a DAG
>
> ```
> ;; Find a path from origine to destiation in a DAG
> ;; find-path: Sym Sym DAG -> (listof Sym)
> (define (find-path orig dest g)
>  (cond
>    [(symbol=? orig dest) (list dest)]
>    [else
>      (local [
>        (define nbrs (neighbours orig g))
>        (define path (find-path/list nbrs dest g))]
>      (cond
>       [(empty? path) empty]
>       [else (cons orig path)]))]))
>
> ;; Find a path from a list of nodes to dest in a DAG,
> ;; producing empty if no path exists
> ;; find-path/list: (listof Sym) Sym DAG -> (listof Sym)
> (define (find-path/list nbrs dest g)
>  (cond
>    [(empty? nbrs) empty]
>    [else
>      (local [(define path (find-path (first nbrs) dest g))]
>        (cond
>          [(empty? path)(find-path/list (rest nbrs) dest g)]
>          [else path]))]))
> ```
>
> This Algo always terminates

Implicit Backtracking

- The find-path functions for implicit backtracking look very similar to those we have developed.
- The neighbours function must now generate the set of neighbours of a node based on some description of that node (e.g. the placement of pieces in a game).
- This allows backtracking in situations where it would be inefficient to generate and store the entire graph as data.
- Backtracking in implicit graphs forms the basis of some artificial intelligence programs, though they generally add heuristics to determine which neighbour to explore first, or which ones to skip because they appear unpromising.

Cycle in Graph

🐋 Info — Path Finding Algo

We add an accumulator `seen` to store visited node to avoid infinte loop

```
;; find-path/acc: Sym Sym Graph (listof Sym)
;; -> (listof Sym)

(define (find-path/acc orig dest g seen)
 (cond
   [(symbol=? orig dest) (list dest)]
   [else
     (local
       [(define nbrs (neighbours orig g))
        (define path (find-path/list nbrs dest g (cons orig seen)))]
     (cond
       [(empty? path) empty]
       [else (cons orig path)]))]))


;; find-path/list: (listof Sym) Sym Graph (listof Sym) -> (listof Sym)
(define (find-path/list nbrs dest g seen)
 (cond
   [(empty? nbrs) empty]
   [(in? (first nbrs) seen) (find-path/list (rest nbrs) dest g seen)]
   [else
     (local
       [(define path (find-path/acc (first nbrs) dest g seen))]
     (cond
       [(empty? path) (find-path/list (rest nbrs) dest g seen)]
       [else path]))]))


;; Find a path from orig to dest in a Graph,
;; producing empty if no path exists
;; find-path: Sym Sym Graph -> (listof Sym)

(define (find-path orig dest g)
 (find-path/acc orig dest g empty))
```

Time complexity: $O(a^n)$