Java II
Graph Algorithms II

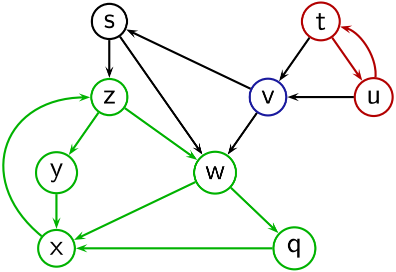
Bernd Kiefer

Deutsches Forschungszentrum für künstliche Intelligenz

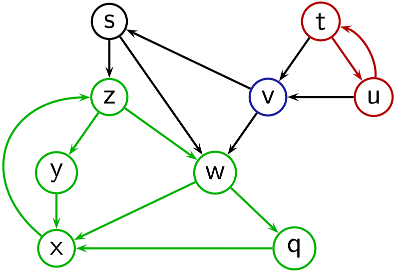
Strongly Connected Components

- Definition: a SCC of a directed graph is the maximal set U of vertices, such that for all $u, v \in U : u \rightarrow v \wedge v \rightarrow u$
- SCCs consist of connected cycles of the graph
- Vertices not in any cycle constitute their own SCC
- The SCCs form a total partition of the graph
- The *component graph*, where the SCCs are replaced by vertices, is acyclic
- Many algorithms are easier to solve on acyclic graphs
  - Run the algorithm on the harder, but smaller SCCs
  - Combine the results on the acyclic component graph
  - a special kind of divide and conquer

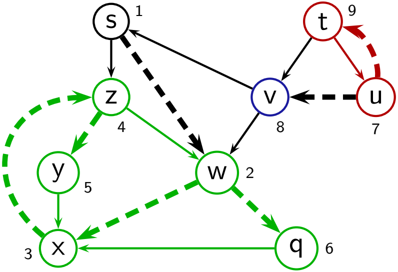
Strongly Connected Components



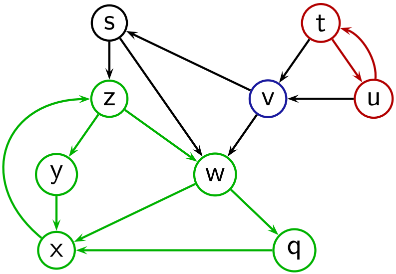
Strongly Connected Components



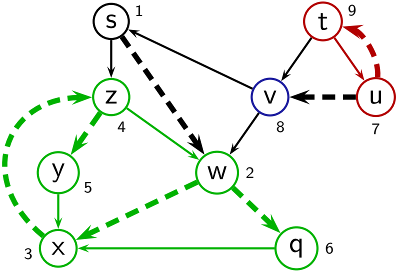
DFS starting at s then u



Strongly Connected Components



DFS starting at s then u



- All nodes of a SCC will be visited in the same DFS
- All vertices of an SCC are connected by tree edges
- There must be a *highest entry node*
- It is the vertex with the lowest discovery time in the SCC

Tarjan's Algorithm

- Crucial observation: Root Property
  - store, for every vertex, the lowest discovery time of an active vertex reachable from it in the DFS tree $\mapsto low(v)$
  - a vertex with $d(v) == low(v)$ is the root of an SCC
  - the SCC consists of the root and all vertices that
    - are on the DFS tree below the root and
    - don't belong to another SCC

      $\longrightarrow$ we can use a stack to collect these vertices
  - $low(v) < d(v)$ can only occur when using
    - back edges
    - cross edges pointing to a vertex still on the stack (active SCC)
  - vertices not on the stack are not considered because they belong to an already finished SCC in another DFS tree branch
  - Pop the SCC vertices when reaching the root

Tarjan's SCC Algorithm

proc *StronglyConnectedComponents*$(G) \equiv$
  for $v \in \mathcal{V}$ **do** $d(v) = 0;\ low(v) = 0;$
  for $v \in \mathcal{V}$ **do if** $d(v) == 0$ **then** $findSCC(v)$

proc *findSCC*$(v) \equiv$
  $low(v) = d(v) = ++time;\ S.push(v);$
  for $e = (v, u) \in \mathcal{E}$ **do**
    if $d(u) == 0$ **then** // u not visited: recurse
      $findSCC(u);\ low(v) = min(low(v), low(u))$
    elsif u is in S **then** $low(v) = min(low(v), low(u))$
  if $d(v) == low(v)$ **then**
    while $S.top() \neq v$ **do** $u = S.pop();$
    $S.pop()$ // pop root

To check efficiently if u is in S, use an additional `boolean`

Search in Weighted Graphs

- Many applications require weights attached to the edges e.g., the transition probabilities
- Goal: find the shortest path
- We will look at single-source shortest path with nonnegative weights
  - The Bellman-Ford algorithm works with negative weights, too
  - For graphs with negative cycles, the shortest path is not well defined
- First: Dijkstra's algorithm for SSSP with nonnegative weights
- Generalization: A* search with a heuristic function

Dijkstra's SSSP

- Algorithm relies on the *triangle equation*: $dist(u, w) + weight(w, v) \geq dist(u, v)$ for all $u, v, w \in \mathcal{V}$
- Initially:
  - set the distances for all nodes to $+\infty$, except for the source node s to zero
  - mark all nodes as not optimized
- While there are nodes not yet optimized:
  - take the unoptimized node u with the smallest $dist(u)$
  - check for all neighbours v if the triangle equation is violated, that is: $dist(u) + weight(u, v) < dist(v)$
  - if so, correct $dist(v)$ and store u as predecessor of v

Dijkstra's SSSP II

```
1 proc Dijkstra-SSPP(s, G) ≡
2   for v ∈ 𝒱 do dist(v) = +∞; predecessor(v) = undef; Q.add(v)
3   dist(s) = 0
4   while Q ≠ ∅ do
5           u = Q.extract_min()
6           for (u, v) ∈ ℰ do
7               alt = dist(u) + weight(u, v)
8               if alt < dist(v) then
9                  dist(v) = alt; predecessor(v) = u
```

- Finally, the predecessor chain can be followed backwards from any node for the shortest path to s
- The algorithm can be stopped in line 5 if u is the desired target node

Data Structure for Q

- Q must support the operations
  - add element
  - extract_min : get and remove the element with the lowest key
  - lower_key : lower the key of an arbitrary element
- `java.util.PriorityQueue` supports the first two efficiently

Data Structure for Q

- Q must support the operations
  - add element
  - extract_min : get and remove the element with the lowest key
  - lower_key : lower the key of an arbitrary element
- `java.util.PriorityQueue` supports the first two efficiently
- *BUT:* lower_key can only be implemented using: `remove(v)+add(v)`, which means $O(n) + O(\lg(n))$
- To avoid the search in `remove(v)`, relate the *elements* efficiently to the *buckets* of the priority queue
- To do so, we need to use a homemade priority queue

Binary Heaps

A heap is a *binary tree* that

- is complete: each level of the tree is completely filled, except for the leaf level, which is filled from left to right
- satisfies the *heap order property*: the data stored in a tree node is smaller (greater) than any in its children

A* Search

- If the search space is very big (as in most AI complete problems), Dijkstra's algorithm may be too expensive
- Use additional information to guide the search, if available
- This will only affect the average time for finding the goal
- Incrementally explore all paths until the optimal path is found:
  - The solution is sound and complete
  - Because of the additional bookkeeping, it can get worse than the plain algorithm, but will behave better in practices

A* Search: Example

- Get from Place s to t using the map of a city
  - Vertices: crossings
  - Edges: connecting roads (eventually one-way)
  - Weights: Length of the road between two crossings
  - If at crossing x, we know the $dist(x)$ already traveled
  - In addition, we have an estimate for the rest: the air-line distance between x and the target t
- Instead of using $dist(x)$ (Dijkstra), use $dist(x) + airline(x, t)$ as weight for the priority queue
- If the remaining cost are never *overestimated*, the heuristic is *admissible* and the optimum will be found
- Dijkstra is a special A*, with the rest cost estimate zero

Collections: List Implementations

- `LinkedList`
  - \+ Constant insertion, deletion of any element, linear merge, efficient stack *and* queue
  - − `get(i), set(i, E)` is linear ($O(n)$)
  - − Space overhead compared to `ArrayList`
- `ArrayList`
  - \+ Constant (almost) insertion, deletion at the end, efficient stack, otherwise linear
  - \+ `get(i), set(i, E)` is constant
  - \+ uses less space than LinkedList
  - − when using many small `ArrayLists`, make sure to create it with a reasonable initial capacity
- All Sorting on `List` should be $O(n \lg(n))$

Set/Map Implementations

- `HashSet`, `HashMap`
  - \+ add, delete, contains, size in amortized constant time
  - \+ linear union, intersection
  - − iteration performance depends on load factor
- `TreeSet`, `TreeMap`
  - add, delete, contains are $O(lg(n))$
  - \+ ordered iteration over the elements
  - \+ using `SortedSet`, it is possible to get, e.g., the next bigger element contained in the set in $O(lg(n))$: `ts.tailSet(elt).first()`

Special Collections

- `BitSet`
  - \+ extremely compact representation of a set
  - \+ very fast union, intersection, etc. using logical bit operations
  - − requires numeric indices
- `Priority Queue`
  - Queue where the elements are sorted according to some `Comparator`
  - asymptotic efficiency similar to `TreeSet`: add/remove is O(lg(n))
  - \+ uses less space than `TreeSet`
  - \+ should be somewhat faster in practice