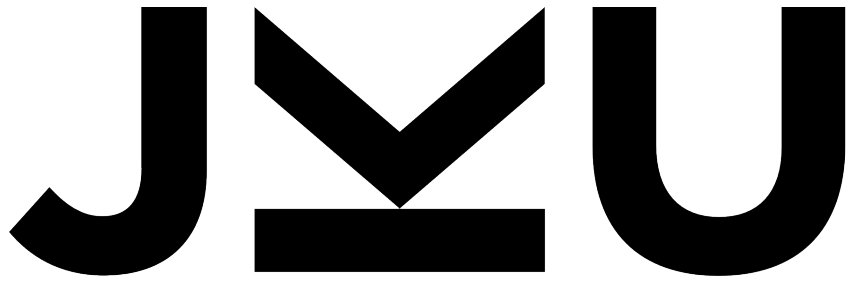


JOHANNES KEPLER
UNIVERSITY LINZ

UE ARTIFICIAL INTELLIGENCE

344.021, 344.022, 344.023, 344.057 (WS 2018)

Andreas Arzt Verena Haunschmid Rainer Kelz

2018/10/12

Institute of Computational Perception

DLDFS WITH CYCLE AVOIDANCE

DLDFS WITH CYCLE AVOIDANCE

let's consider a depth limited depth first search (DLDFS)

- branching factor b, depth limit d
- runtime: $O(b^d)$, since we expand all nodes
- space: $\mathbf{O(b \cdot d)}$ (we only keep the current path in memory)
- a "closed set" would bump space complexity to $O(b^d)$!
- it would have the **same space requirements as BFS**, but **would not even be optimal**! we could have used BFS in the first place ...

HOW CAN WE STILL USE DLDFS?

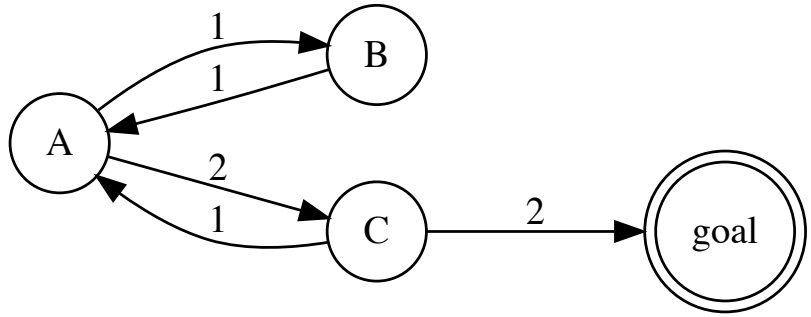
- we cannot remember **all** the nodes we expanded
- but we want to avoid at least **some** cycles
- what is the best we can do ?

HOW CAN WE STILL USE DLDFS?

- we cannot remember **all** the nodes we expanded
- but we want to avoid at least **some** cycles
- what is the best we can do ?
- the best we can do to keep the same space complexity, and avoid unnecessary work, is to avoid expanding nodes **on the current path** again!
- we need to **check** whether an expanded node is already **on the current path**

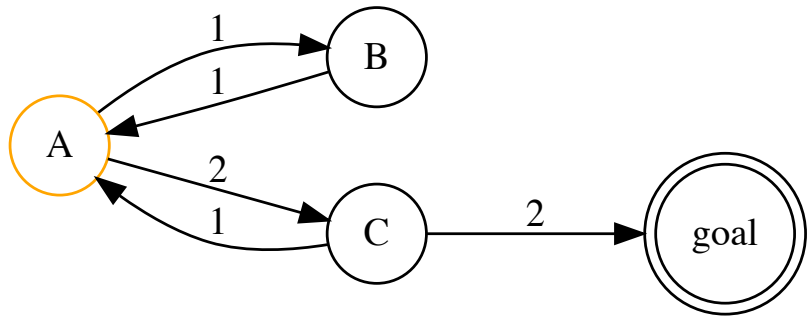
PROBLEM - NO CYCLE AVOIDANCE

consider this example graph, numbers on edges denote expansion order



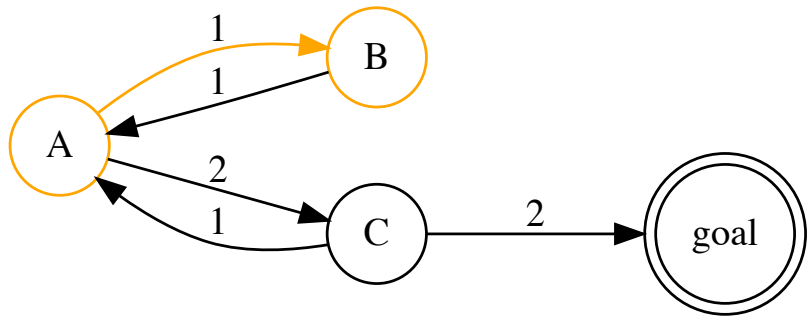
PROBLEM - NO CYCLE AVOIDANCE

we start in node A



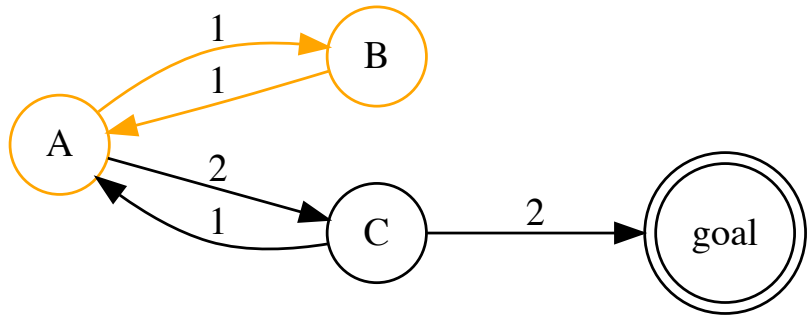
PROBLEM - NO CYCLE AVOIDANCE

we expand B first



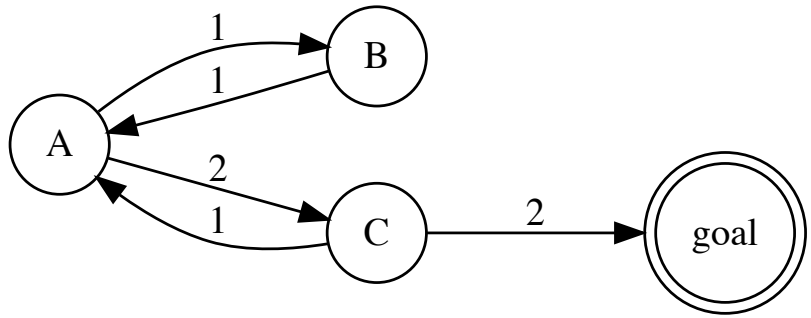
PROBLEM - NO CYCLE AVOIDANCE

we expand A again, because we didn't remember anything! this will cycle until the depth limit is reached, and is very wasteful, especially in loopy state spaces.



SOLUTION - WITH CYCLE AVOIDANCE

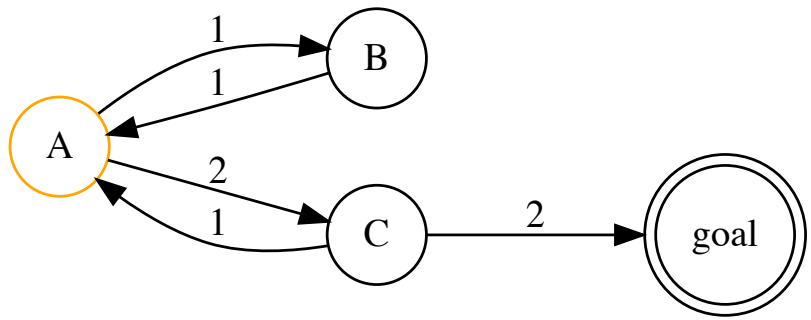
considering the same example graph, numbers on edges denote expansion order, but this time we'll remember what nodes are on the **current path**!



SOLUTION - WITH CYCLE AVOIDANCE

we start at node A, remembering A

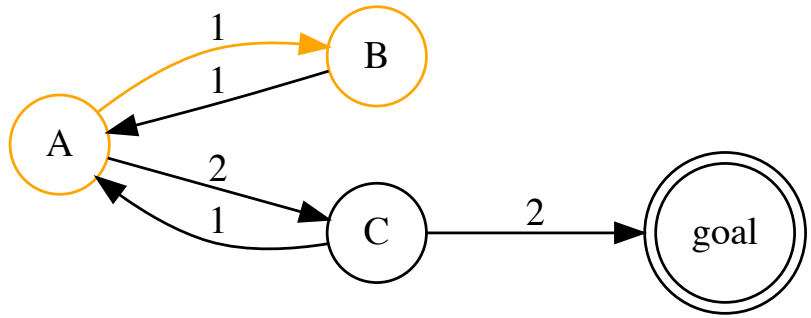
(“remembering” means `push()`-ing it on the stack)



SOLUTION - WITH CYCLE AVOIDANCE

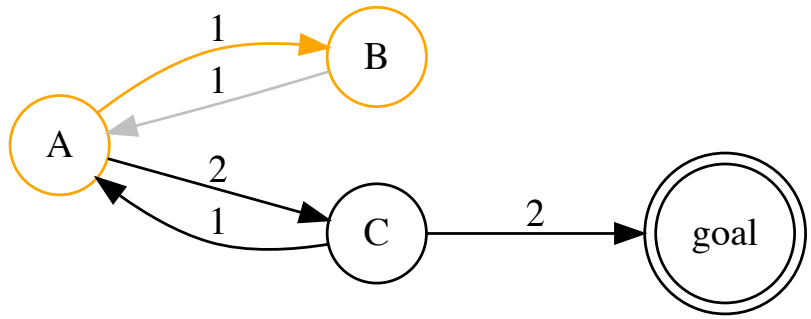
we expand B first, remembering B

(“remembering” means `push()`-ing it on the stack)



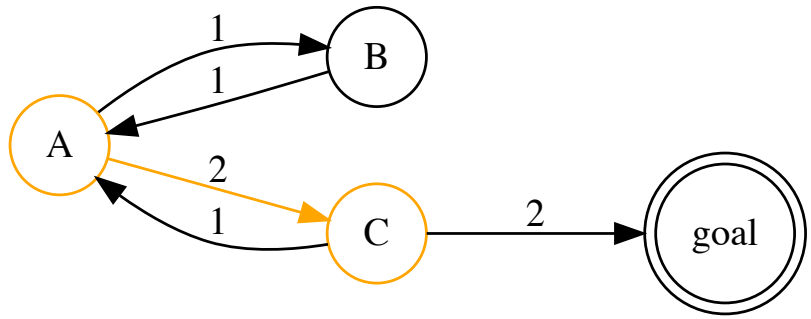
SOLUTION - WITH CYCLE AVOIDANCE

we should expand A again, but we remember it being on the **same path**. this avoids cycling until the depth limit is reached, saving some effort.



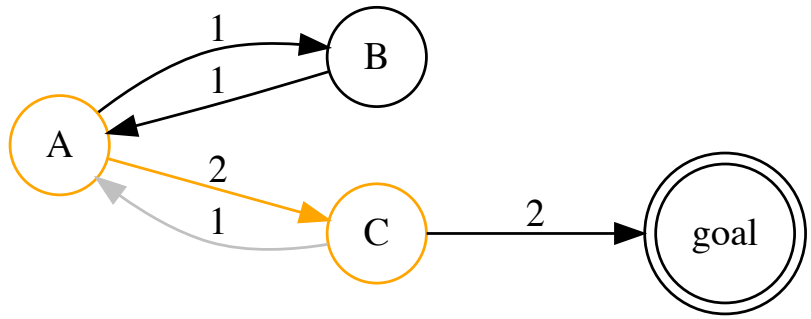
SOLUTION - WITH CYCLE AVOIDANCE

we expand C now, having explored the left-most subtree, **forgetting** the path we are not on anymore! (“forgetting” means `pop()`-ping the element off the stack).



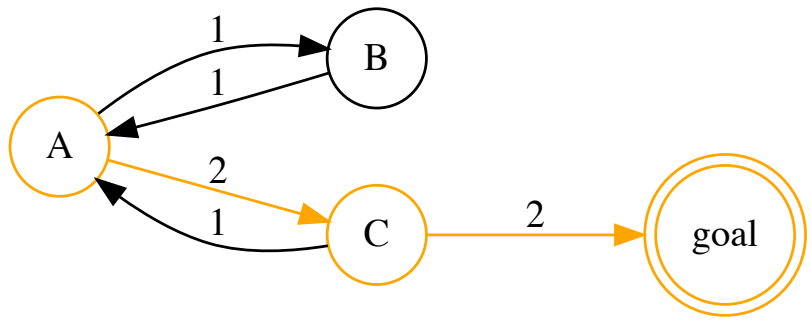
SOLUTION - WITH CYCLE AVOIDANCE

we should expand A again, but we remember it being on the **same path**. this avoids cycling until the depth limit is reached, saving some effort.



SOLUTION - WITH CYCLE AVOIDANCE

we expand the next node, which happens to be the goal node (again, after having explored the left-most subtree)



CONCLUSION I

- **do not use a “closed set”** with (DL)DFS
- instead, only **avoid loops on the current path**
- this is called **self-avoiding walk**

CONCLUSION II

- DLDFS is usually implemented using a stack
  - implicit call-stack in recursive implementations
  - explicit stack in iterative ones (LIFO)
  - of course, your recursive implementation could also keep an explicit stack...
- check if expanded node is somewhere in the stack ("is it somewhere along the path we walked so far?")
  - we need a stack with a fast "`contains`" operation
  - incidentally, there is one such datastructure provided that has the desired behaviour

`at.jku.cp.rau.search.datastructures.StackWithFastContains`

PRIORITY QUEUES

- store elements with an associated priority
- retrieve elements in order of priority
- `retrieve` operation in $O(\log N)$ yielding the element with the highest priority
- please use the Priority Queue implementation provided in the framework
- this implementation is modified to fall back on insertion order in case of tied priorities
- this **makes** the **behaviour** of your search algorithms **comparable** to ours

`at.jku.cp.rau.search.datastructures.StablePriorityQueue`