HW1 examples

Source: Andy

Destination: Alice

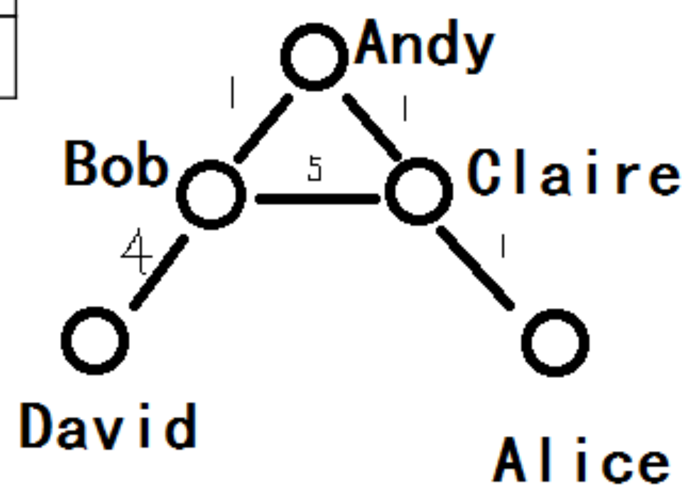
	Andy	Bob	Claire	David	Alice
Andy	0	1	1	0	0
Bob	1	0	5	4	0
Claire	1	5	0	0	1
David	0	4	0	0	0
Alice	0	0	1	0	0

HW1 examples

Source: Andy

Destination: Alice

	Andy	Bob	Claire	David	Alice
Andy	0	1	1	0	0
Bob	1	0	5	4	0
Claire	1	5	0	0	1
David	0	4	0	0	0
Alice	0	0	1	0	0



Issues

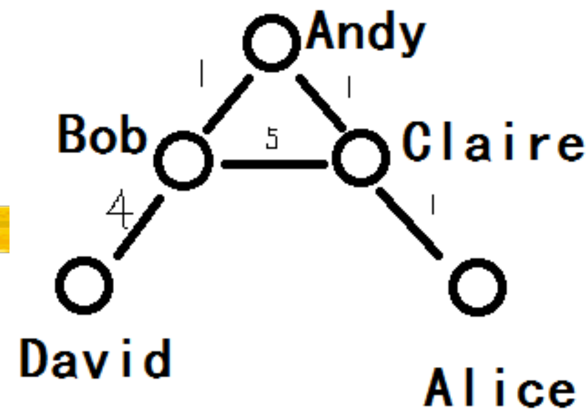
- Reference implementation created by the TAs did not follow the homework specifications exactly.

- "standard" algorithms are usually under-determined, e.g., they do not specify ordering – hence they may not be directly applicable.

- "pure" algorithms, e.g., BFS strictly uses a FIFO queue, are under-determined – most likely those will not work. E.g., your BFS answer could use a FIFO queue, plus some loop detection logic, plus some extra logic to enforce the alphabetical popping.

- "equivalent tricks", e.g., "to achieve alphabetical popping, use reverse alphabetical pushing in DFS" may not work – use only if you are sure your trick truly is equivalent to the stated specifications.

Breadth-first (BFS) search

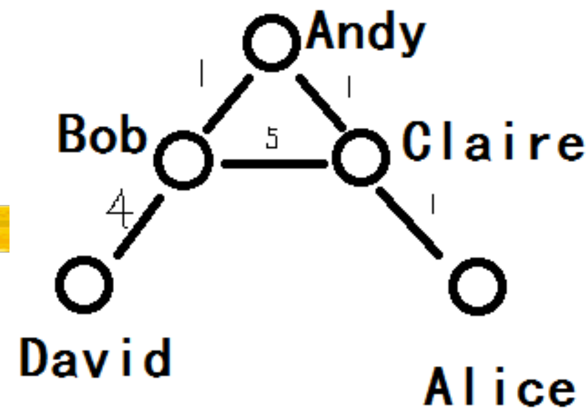


Queue: add successors to queue back;
empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--

Log:

Breadth-first (BFS) search

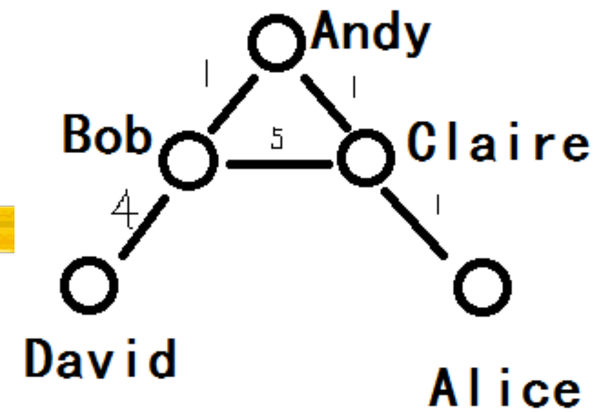


Queue: add successors to queue back;
empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1

Log: Andy

Breadth-first (BFS) search



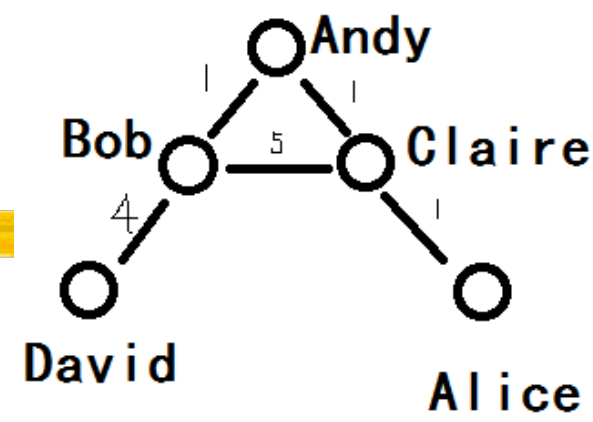
Queue: add successors to queue back; empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1

Note: When the costs of two or more nodes are equal, you need to make sure these nodes are ***popped off the search queue in alphabetical order.*** This will resolve ambiguity and ensure that there is only one correct solution for each problem.

Log: Andy

Breadth-first (BFS) search



Queue: add successors to queue back;
empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1
4	David	2	2	2

Children of #2 (Bob): Andy, Claire, David

Andy: not enqueued (case 3)
Claire: not enqueued (case 2)
David: enqueued (case 1)

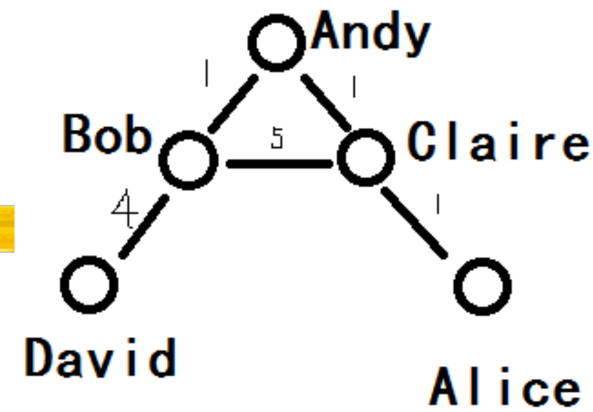
Note 2: Your algorithm should perform ***loop detection***. As studied in lectures 2-4, do not enqueue a child that has a state already visited, unless the child has a better cost than when we previously visited that state (see slides about "A clean robust algorithm" in session02-04.pptx lecture slides).

Log: Andy - Bob

A Clean Robust Algorithm

```
[... see previous slide ...]
          children ← Expand(currnode, Operators[problem])
          while children not empty
               child ← Remove-Front(children)
Case 1         if no node in open or closed has child's state
                    open ← Queuing-Fn(open, child)
Case 2         else if there exists node in open that has child's state
                    if PathCost(child) < PathCost(node)
                         open ← Delete-Node(open, node)
                         open ← Queuing-Fn(open, child)
Case 3         else if there exists node in closed that has child's state
                    if PathCost(child) < PathCost(node)
                         closed ← Delete-Node(closed, node)
                         open ← Queuing-Fn(open, child)
          end
[... see previous slide ...]
```

Breadth-first (BFS) search

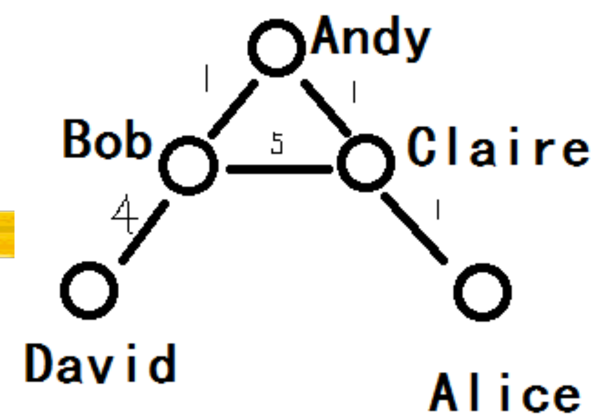


Queue: add successors to queue back; empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1
4	David	2	2	2
5	Alice	2	2	3

Log: Andy - Bob - Claire

Breadth-first (BFS) search

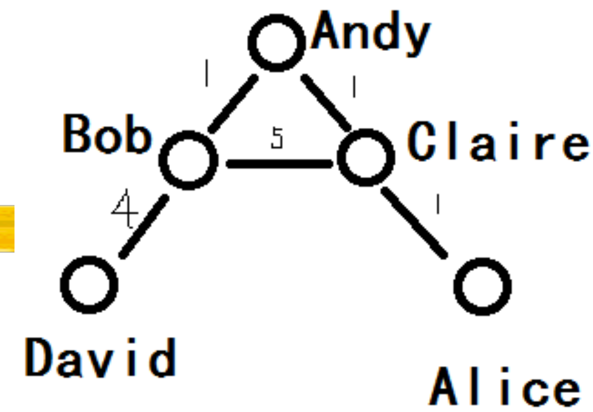


Queue: add successors to queue back; empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1
4	David	2	2	2
5	Alice	2	2	3

Log: Andy - Bob - Claire

Breadth-first (BFS) search



Queue: add successors to queue back; empty queue from front (top)

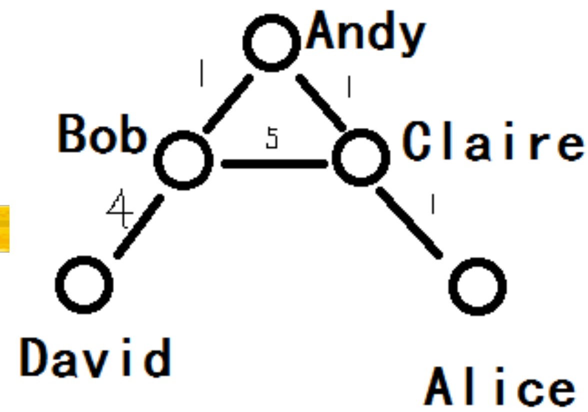
#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1
4	David	2	2	2
5	Alice	2	2	3

To get the path: backtrack from solution node up the chain of parent nodes

Path: … (#5) Alice

Log: Andy - Bob - Claire - Alice

Breadth-first (BFS) search



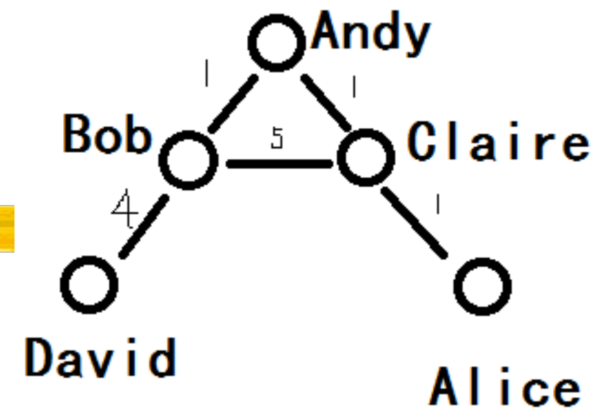
Queue: add successors to queue back; empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1
4	David	2	2	2
5	Alice	2	2	3

Path: … (#3) Claire - (#5) Alice

Log: Andy - Bob - Claire - Alice

Breadth-first (BFS) search



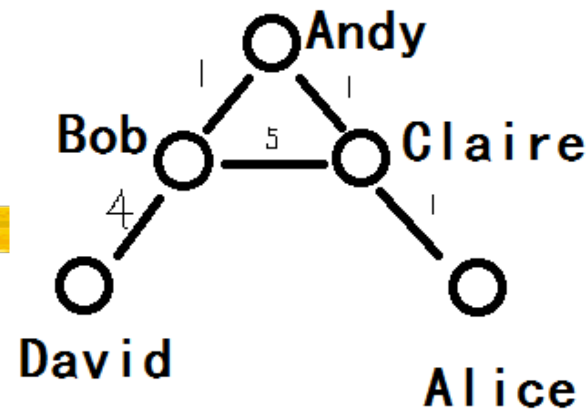
Queue: add successors to queue back;
empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1
4	David	2	2	2
5	Alice	2	2	3

Path: (#1) Andy - (#3) Claire - (#5) Alice

Log: Andy - Bob - Claire - Alice

Depth-first (DFS) search

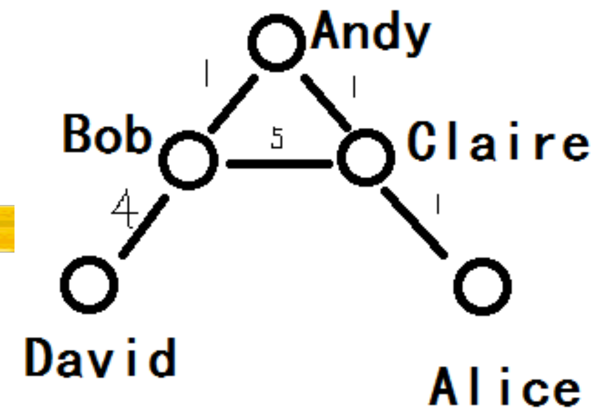


Queue: add successors to queue front; empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--

Log:

Depth-first (DFS) search

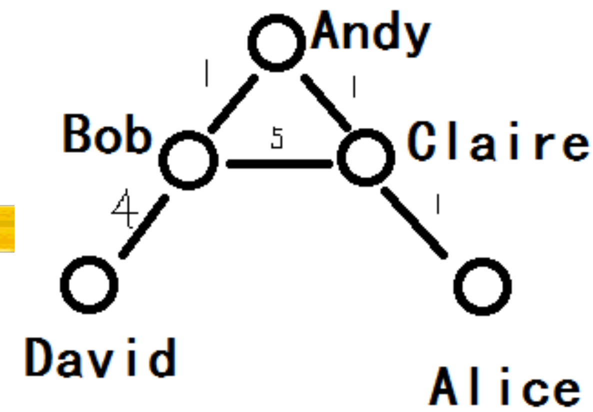


Queue: add successors to queue front;
empty queue from front (top)

#	state	depth	path cost	parent #
2	Bob	1	1	1
3	Claire	1	1	1
1	Andy	0	0	--

Log: Andy

Depth-first (DFS) search

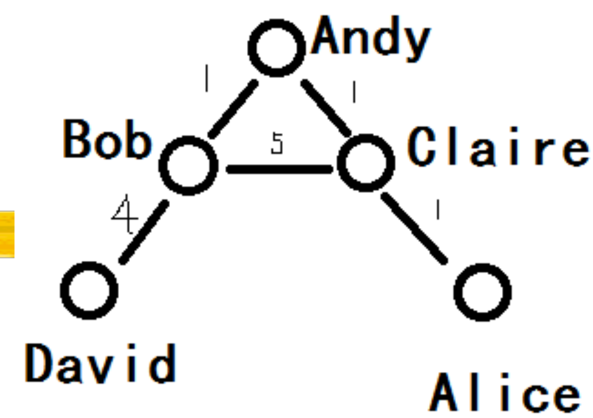


Queue: add successors to queue front; empty queue from front (top)

#	state	depth	path cost	parent #
2	Bob	1	1	1
3	Claire	1	1	1
1	Andy	0	0	--

Log: Andy

Depth-first (DFS) search



Queue: add successors to queue front;
empty queue from front (top)

#	state	depth	path cost	parent #
4	David	2	2	2
2	Bob	1	1	1
3	Claire	1	1	1
1	Andy	0	0	--

Enqueue Claire again? No because of loop detection rule
(node #3 in open queue already has state Claire)

Note 2: Your algorithm should perform ***loop detection***. As studied in lectures 2-4, do not enqueue a child that has a state already visited, unless the child has a better cost than when we previously visited that state (see slides about "A clean robust algorithm" in session02-04.pptx lecture slides).

Log: Andy - Bob

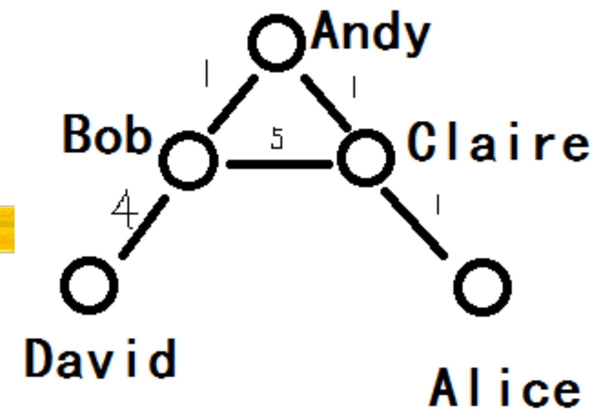
A Clean Robust Algorithm

[… see previous slide …]

```
        children ← Expand(currnode, Operators[problem])
        while children not empty
                child ← Remove-Front(children)
                if no node in open or closed has child's state
                        open ← Queuing-Fn(open, child)
                else if there exists node in open that has child's state
                        if PathCost(child) < PathCost(node)
                                open ← Delete-Node(open, node)
                                open ← Queuing-Fn(open, child)
                else if there exists node in closed that has child's state
                        if PathCost(child) < PathCost(node)
                                closed ← Delete-Node(closed, node)
                                open ← Queuing-Fn(open, child)
        end
```

[… see previous slide …]

Depth-first (DFS) search

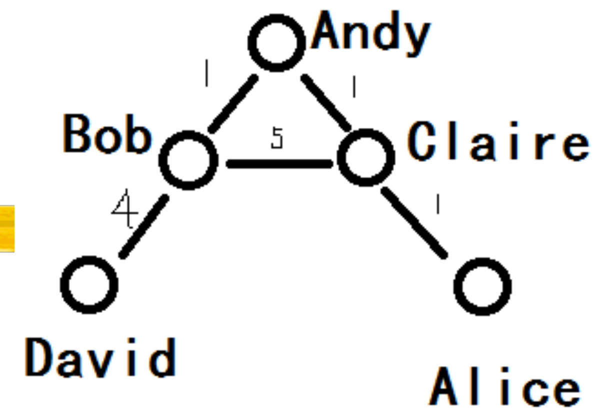


Queue: add successors to queue front;
empty queue from front (top)

#	state	depth	path cost	parent #
4	David	2	2	2
2	Bob	1	1	1
3	Claire	1	1	1
1	Andy	0	0	--

Log: Andy - Bob

Depth-first (DFS) search

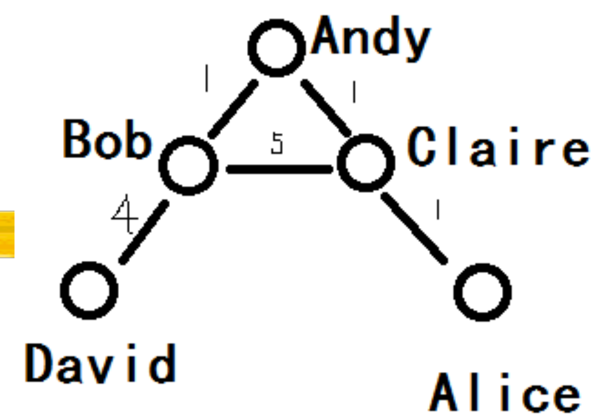


Queue: add successors to queue front; empty queue from front (top)

#	state	depth	path cost	parent #
4	David	2	2	2
2	Bob	1	1	1
3	Claire	1	1	1
1	Andy	0	0	--

Log: Andy - Bob - David

Depth-first (DFS) search

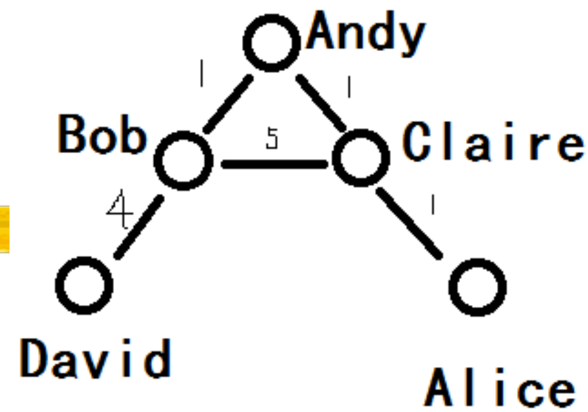


Queue: add successors to queue front; empty queue from front (top)

#	state	depth	path cost	parent #
5	Alice	2	2	3
4	David	2	2	2
2	Bob	1	1	1
3	Claire	1	1	1
1	Andy	0	0	--

Log: Andy - Bob - David - Claire

Depth-first (DFS) search



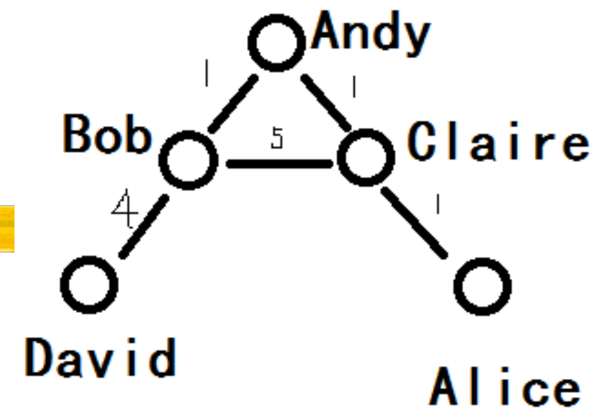
Queue: add successors to queue front;
empty queue from front (top)

#	state	depth	path cost	parent #
5	Alice	2	2	3
4	David	2	2	2
2	Bob	1	1	1
3	Claire	1	1	1
1	Andy	0	0	--

Path: (#1) Andy - (#3) Claire - (#5) Alice

Log: Andy - Bob - David - Claire - Alice

Uniform-cost (UCS) search

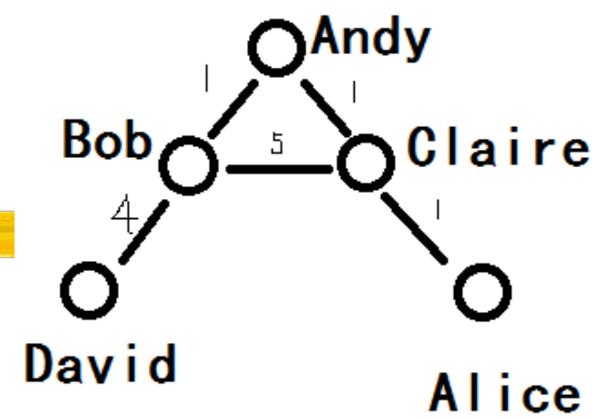


Queue: keep queue sorted by path cost; empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--

Log:

Uniform-cost (UCS) search

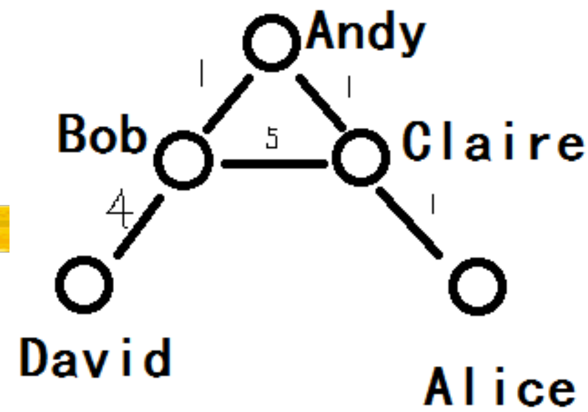


Queue: keep queue sorted by path cost;
empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1

Log: Andy

Uniform-cost (UCS) search

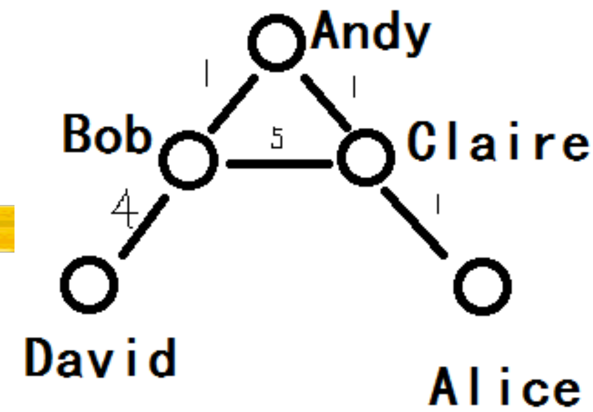


Queue: keep queue sorted by path cost;
empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1
4	David	2	5	2

Log: Andy - Bob

Uniform-cost (UCS) search



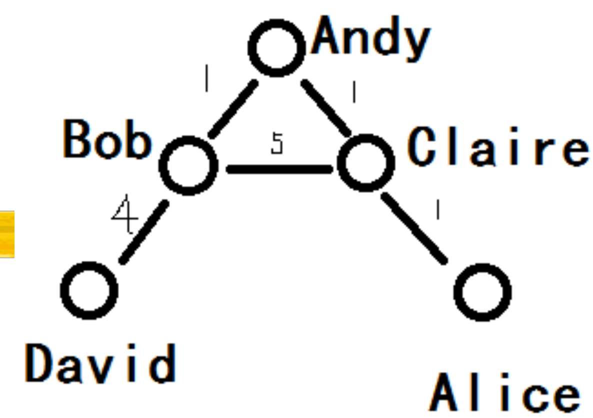
Queue: keep queue sorted by path cost; empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1
5	Alice	2	2	3
4	David	2	5	2

Note the sorting by path cost (#5 goes above #4)

Log: Andy - Bob - Claire

Uniform-cost (UCS) search



Queue: keep queue sorted by path cost; empty queue from front (top)

#	state	depth	path cost	parent #
1	Andy	0	0	--
2	Bob	1	1	1
3	Claire	1	1	1
5	Alice	2	2	3
4	David	2	5	2

Path: (#1) Andy - (#3) Claire - (#5) Alice

Log: Andy - Bob - Claire - Alice

Summary

	Prof Itti	Original HW1	HW1 Erratum
BFS	Log: Andy - Bob - Claire - Alice Path: Andy - Claire - Alice	Log: Andy-Bob-Claire-David-Alice Path: Andy-Claire-Alice	
DFS	Log: Andy - Bob - David - Claire - Alice Path: Andy - Claire - Alice	Log: Andy-Bob-David-Claire-Alice Path: Andy-Claire-Alice	Log: Andy-Bob-Claire-Alice Path: Andy-Bob-Claire-Alice
UCS	Log: Andy - Bob - Claire - Alice Path: Andy - Claire - Alice	Log: Andy-Bob-Claire-Alice Path: Andy-Claire-Alice	

Summary

	Prof Itti	Original HW1	HW1 Erratum
BFS	Log: Andy - Bob - Claire - Alice Path: Andy - Claire - Alice	Log: Andy-Bob-Claire-~~David~~-Alice Path: Andy-Claire-Alice	
DFS	Log: Andy - Bob - David - Claire - Alice Path: Andy - Claire - Alice	Log: Andy-Bob-David-Claire-Alice Path: Andy-Claire-Alice	~~Log: Andy-Bob-Claire-Alice~~ ~~Path: Andy-Bob-Claire-Alice~~
UCS	Log: Andy - Bob - Claire - Alice Path: Andy - Claire - Alice	Log: Andy-Bob-Claire-Alice Path: Andy-Claire-Alice	

Recommendation

- As already recommended earlier, please make sure your algorithm complies with the specifications given in the HW handout. Following the specs overrides following the examples.

- "standard" algorithms are usually under-determined, e.g., they do not specify ordering – hence they may not be directly applicable.

- "pure" algorithms, e.g., BFS strictly uses a FIFO queue, are under-determined – most likely those will not work. E.g., your BFS answer could use a FIFO queue, plus some loop detection logic, plus some extra logic to enforce the alphabetical popping.

- "equivalent tricks", e.g., "to achieve alphabetical popping, use reverse alphabetical pushing in DFS" may not work – use only if you are sure your trick truly is equivalent to the stated specifications.

- please do not try to convince the TAs that some standard implementation should give the correct result for this problem – standard implementations usually are not 100% compliant with this specific problem.

- if/where possible ambiguity remains, try your best to comply with the specifications and let us know your thoughts in a README file.