Verification of an Airport Taxiway Path-Finding Algorithm

Siyuan He, Ke Du, Joshua Wilhelm and Jean-Baptiste Jeannin
hesy@umich.edu
madoka@umich.edu

University of Michigan

DASC 2020

Why autonomous taxiing?

- Added safety
  - In 2017, there were no commercial passenger jet fatalities in the world
  - However, about 35 people died on the ground (ramp workers, luggage handlers…)
- Eliminating runway incursions
  - Could be extremely dangerous if not cleared by control
- Reducing pilot’s workload
  - Pilots can focus on preparing the flight
  - They can monitor the autonomous taxiing rather than perform the taxiing
  - They don’t have to worry about the route

Formally Verified Taxiway Path-Finding

- Main contribution: formal verification of the taxiway path-finding algorithm [1]
- Tool: interactive proof assistant, Coq

[1] Zhang, Yuhao, et al. "A software architecture for autonomous taxiing of aircraft." *AIAA Scitech 2020 Forum*. 2020.

Coq: Interactive Theorem Prover

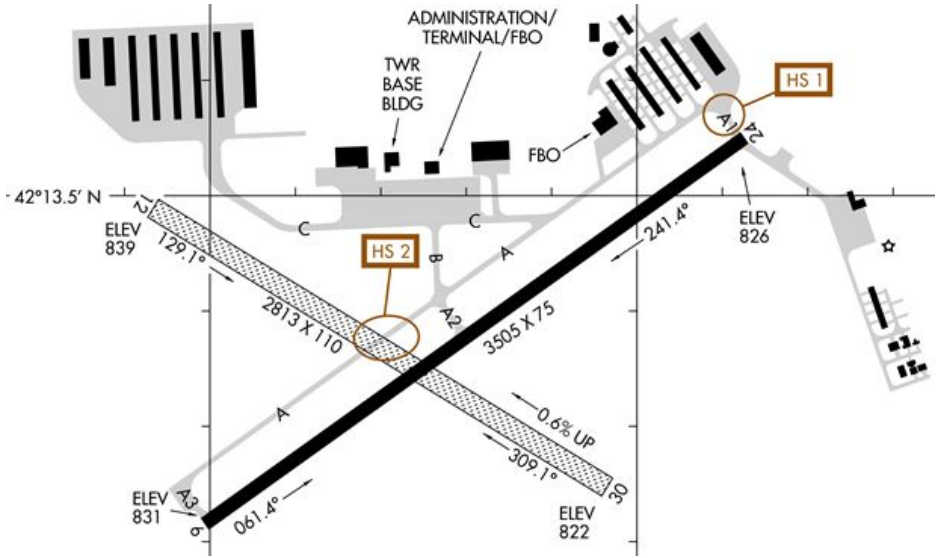
- For verifying software systems, algorithms, mathematical theorems, etc
  - CompCert, CertiKOS
- Work done in Coq:
  - Algorithm implemented
  - Formal Specification for the correctness properties
  - A proof of the specification
- The proof is checked by machine (Coq)

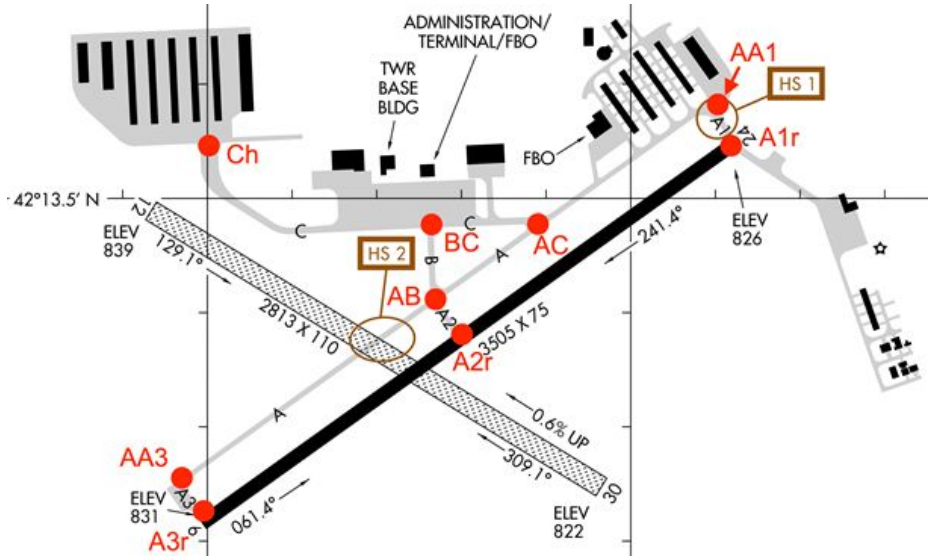
Correctness of a Path Found by the Algorithm

- The path is an actual path in the airport ✓
- The path follows ATC command ✓
- A plane cannot take certain movements (Eg. U-turn)
- The algorithm finds a path, if any (future work)
- ...

Model of the taxiways: a mathematical graph

- FAA diagram of an airport is a map of taxiways
- vertices = intersections, edges = taxiways

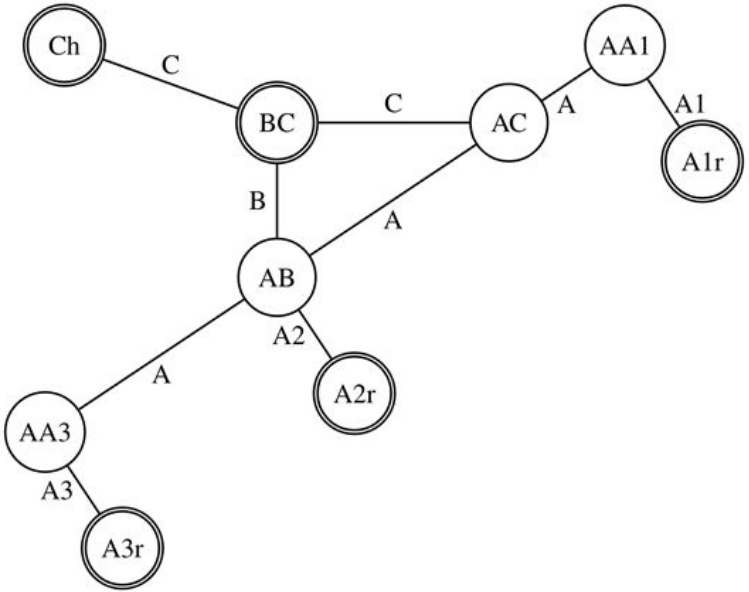


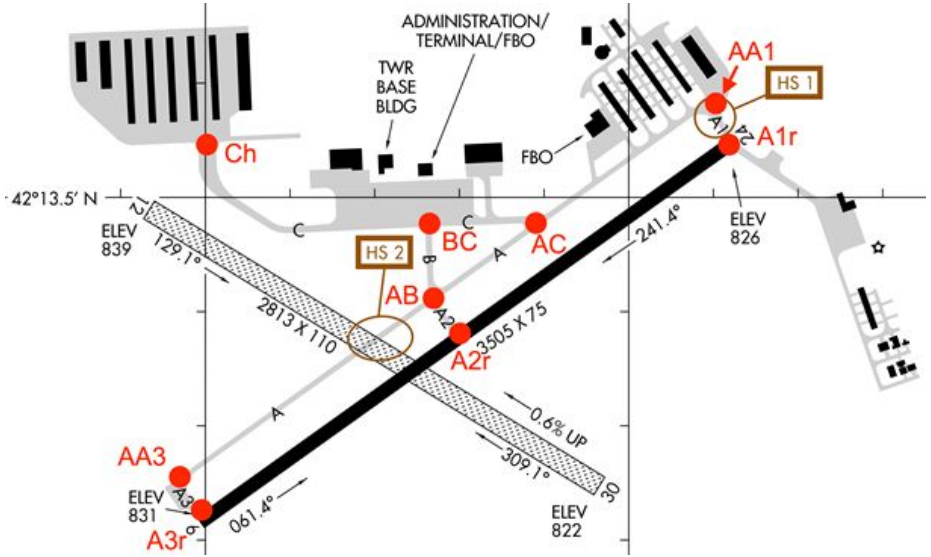


Example: ann arbor airport (KARB)

Model of the taxiways: a mathematical graph

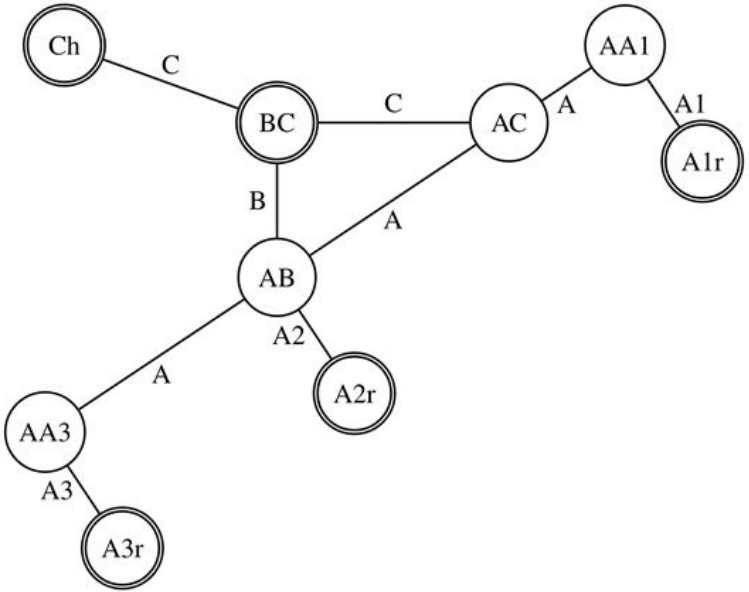
- **Input**: start=*Ch*, end=*A2r*, taxiways=[C,B,A2]
- **Output**: path=[Ch, BC, AB, A2r]

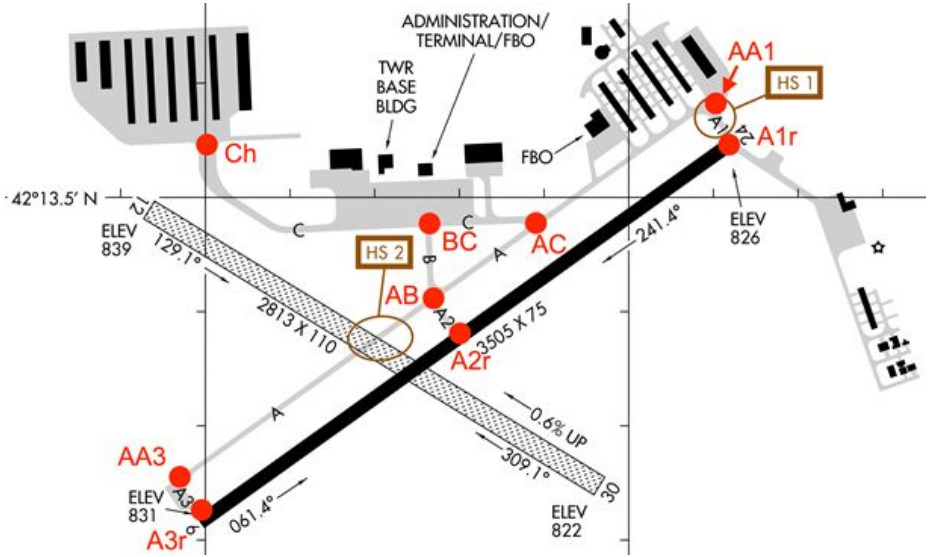




Model of the taxiways: a mathematical graph

- But aircrafts can turn around in this graph, which is impossible in reality





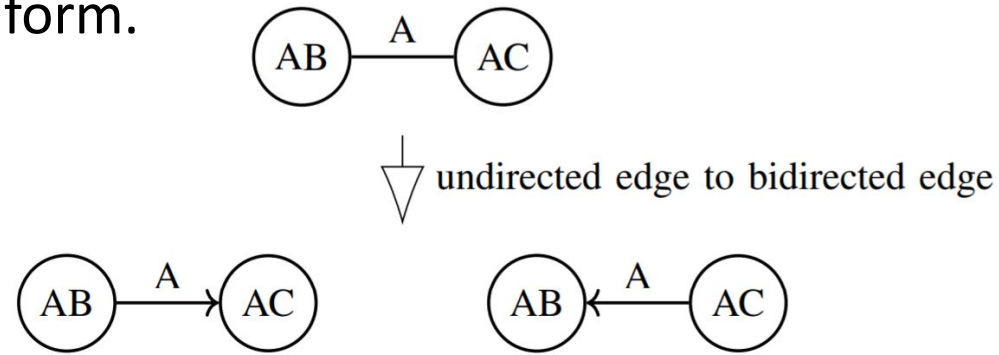
Graph Expansion

Solution:
undirected graph → ***directed expanded graph*** [1]

[1] Zhang, Yuhao, et al. "A software architecture for autonomous taxiing of aircraft." *AIAA Scitech 2020 Forum*. 2020.

Undirected Graph to Bidirected Graph

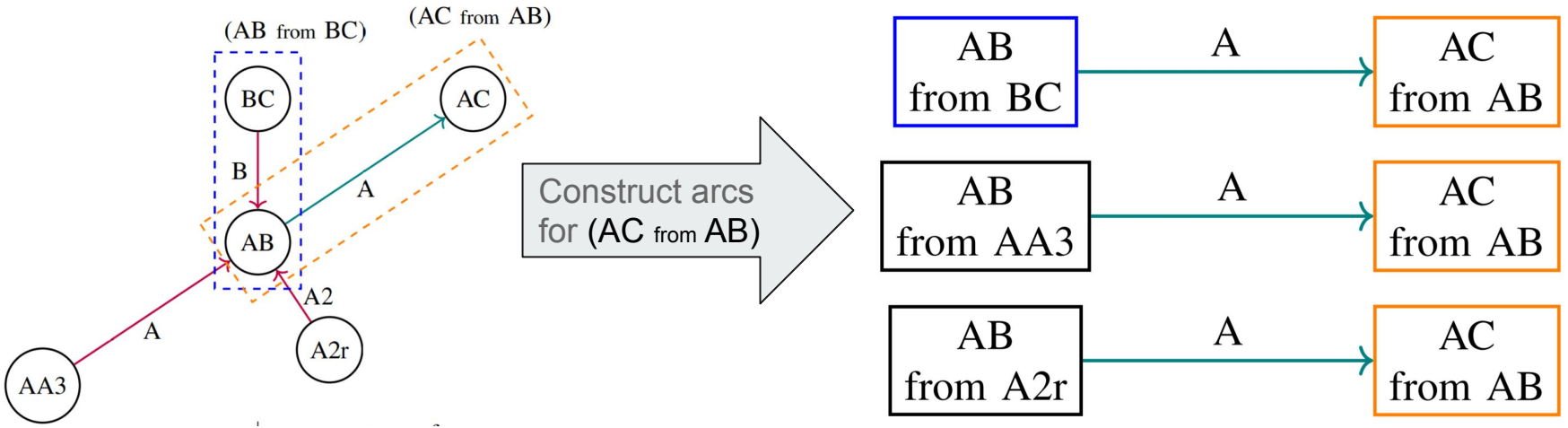
Turn every undirected edge to its equivalent bidirected form.



[1] Zhang, Yuhao, et al. "A software architecture for autonomous taxiing of aircraft." *AIAA Scitech 2020 Forum*. 2020.
[2] Harary, Frank, and Robert Z. Norman. "Some properties of line digraphs." *Rendiconti del Circolo Matematico di Palermo* 9.2 (1960): 161-168.

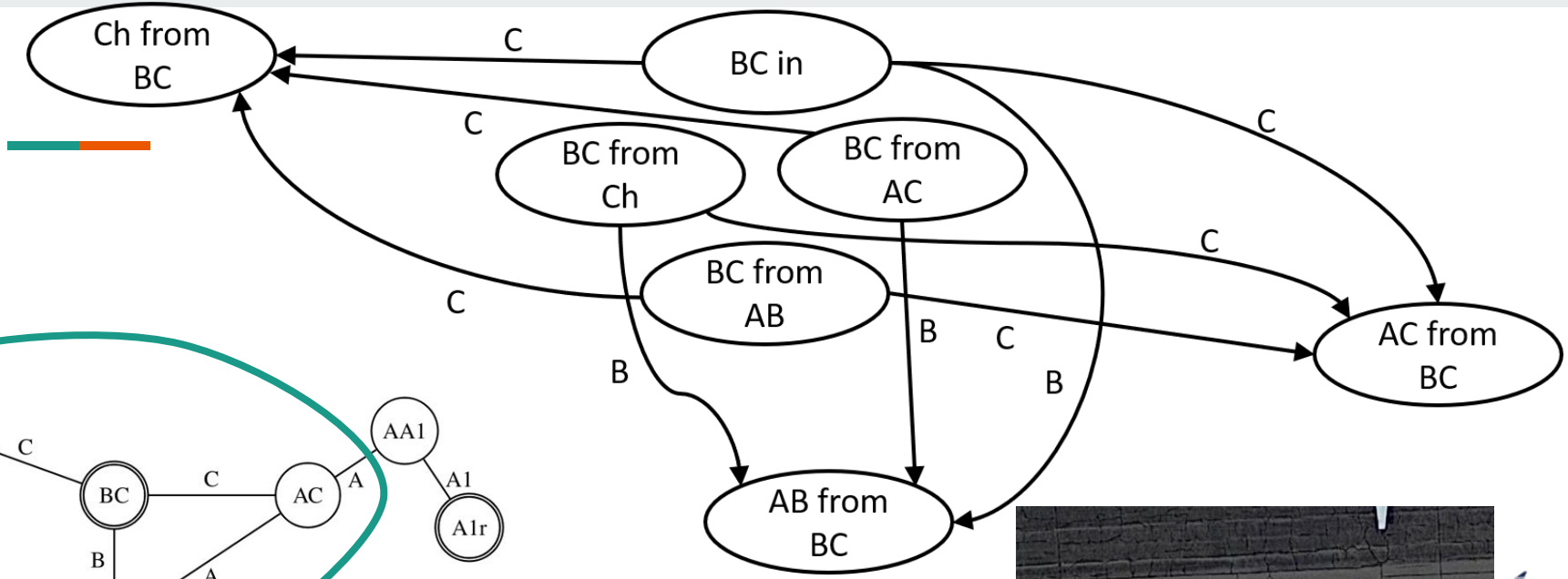
Bidirected graph to Directed Expanded Graph

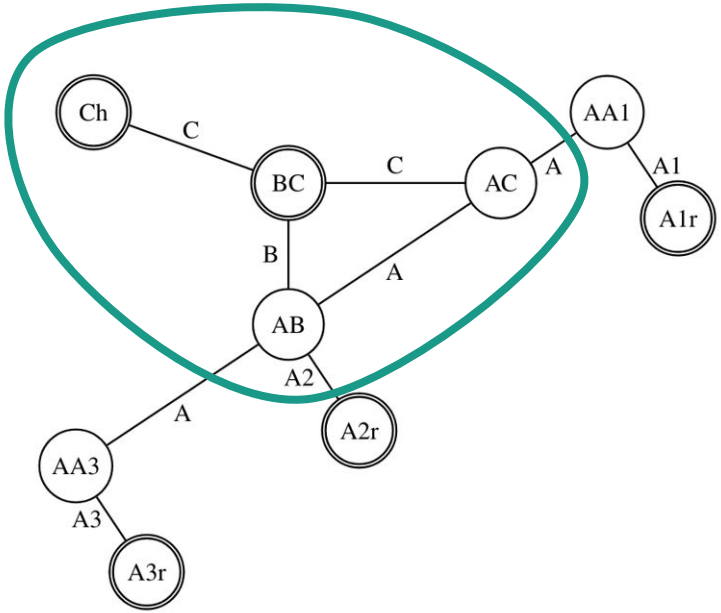
Arc (in directed expanded graph) := (previous_edge, current_edge)

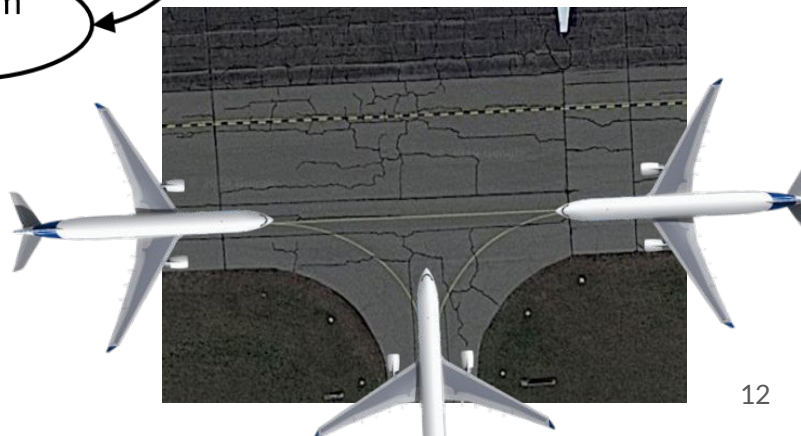


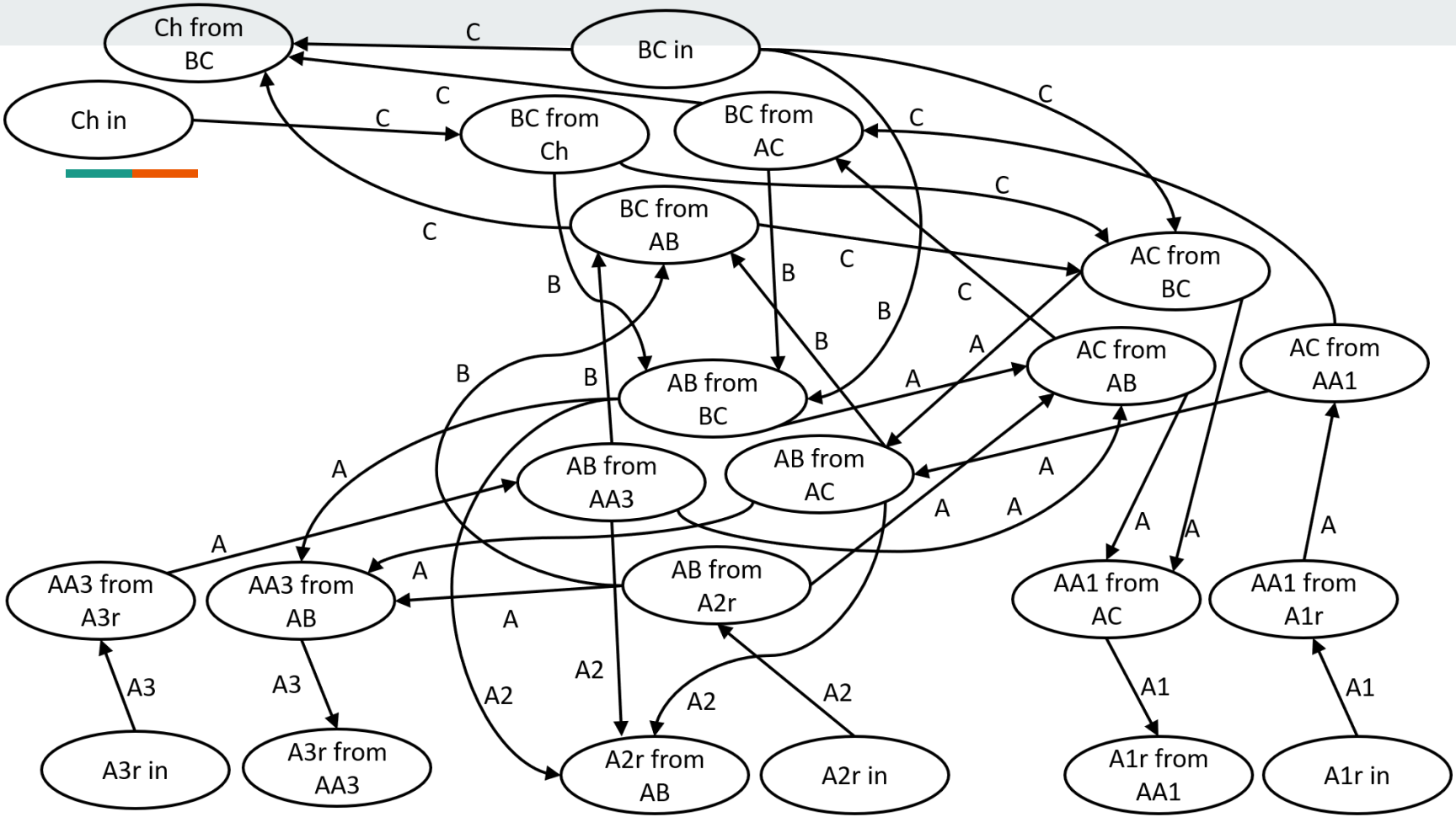
Subgraph of the bidirected graph

Three arcs constructed w.r.t. the subgraph

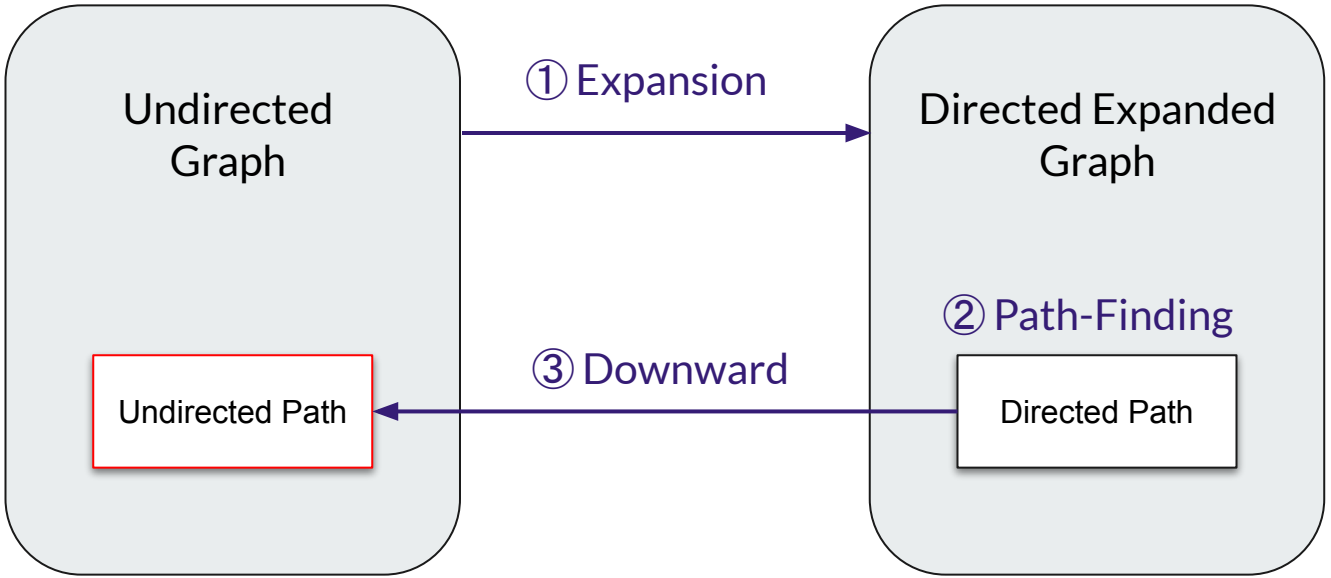

Ch from BC
BC in
BC from Ch
BC from AC
BC from AB
AC from BC
AB from BC
C
B


Ch
BC
AC
AA1
A1
A1r
AB
A2
A2r
AA3
A3
A3r
A
B
C

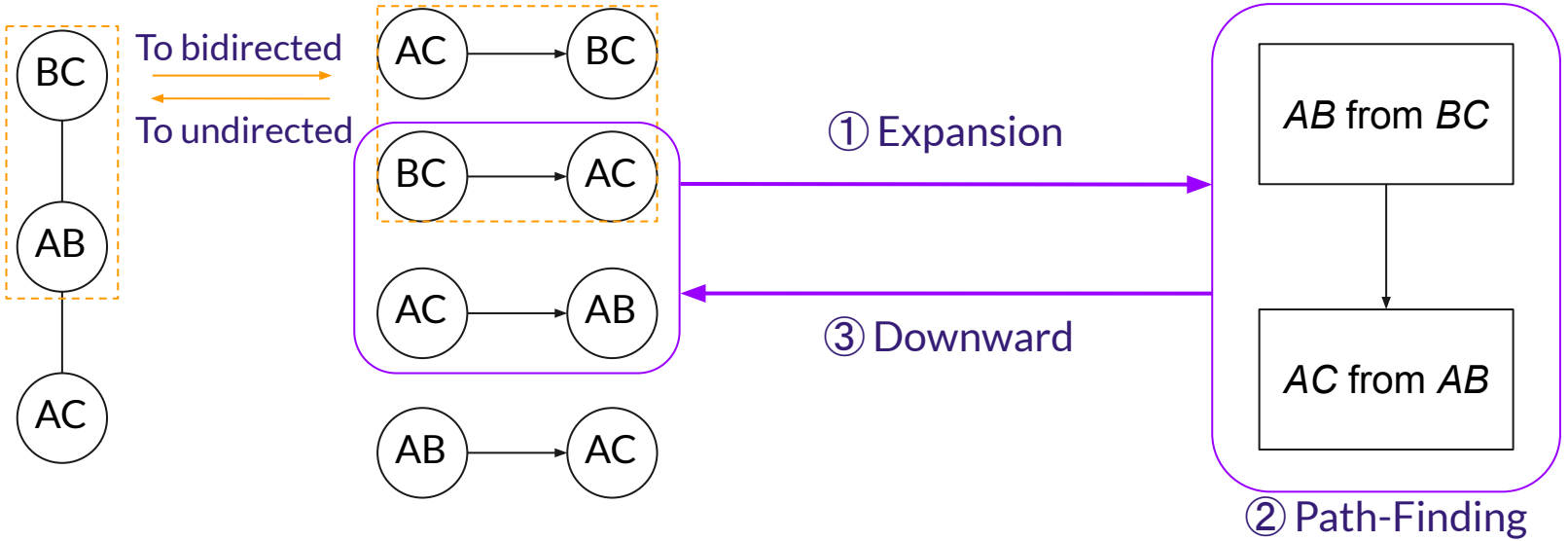



Ch from BC
BC in
Ch in
BC from Ch
BC from AC
BC from AB
AC from BC
AC from AB
AC from AA1
AB from BC
AB from AA3
AB from AC
AB from A2r
AA3 from A3r
AA3 from AB
AA1 from AC
AA1 from A1r
A3r in
A3r from AA3
A2r from AB
A2r in
A1r from AA1
A1r in
C
B
A
A3
A2
A1

Algorithm - Overview



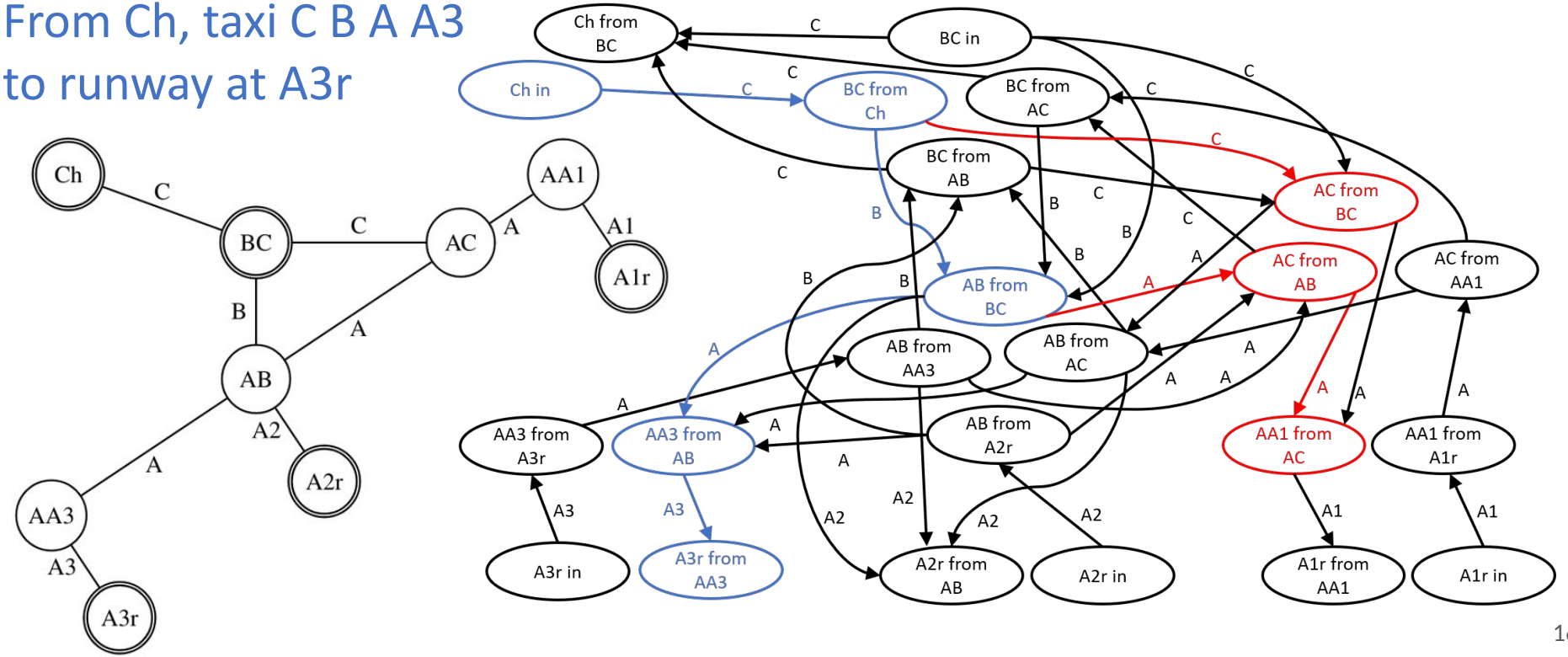
Algorithm - Expansion and Downward



Special Case: If a vertex *V* is connected to *input*, we expand to (*input* from *input* -> V from *input*)

Algorithm - Path-Finding

From Ch, taxi C B A A3 to runway at A3r



Algorithm - State in Path-Finding

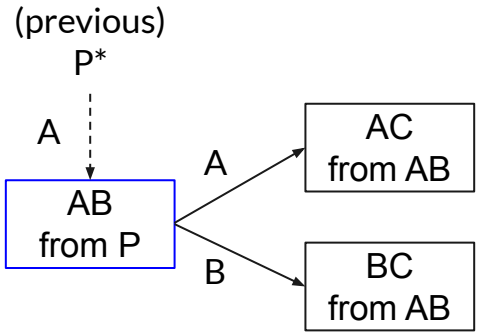
State: The intermediate data structure in path-finding.

A state consists of four necessary information to identify an intermediate status.

- Previous route from start point to current position (reverse order)
- Taxiway name of current position
- Remaining ATC command to go through
- Previous followed ATC command

Able to reconstruct original ATC by combining these elements

Algorithm - Step State in Path-Finding



Step on
current taxiway

Step on
next taxiway

current State 0 (AB from P):

- AB from P <- P* (previous path)
- A
- previous ATC
- B :: future ATC

next State 1 (BC from AB):

- AC from AB <- AB from P <- P*
- A
- previous ATC
- B :: future ATC

next State 2 (BC from AB):

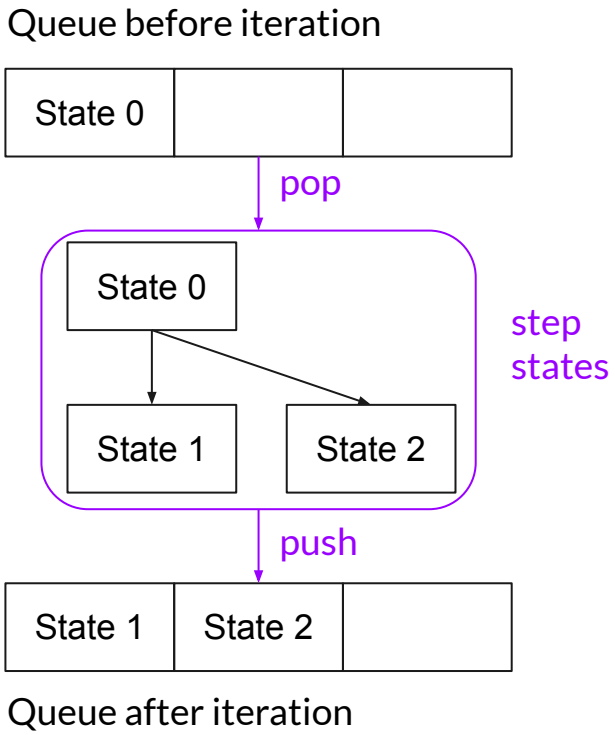
- BC from AB <- AB from P <- P*
- B
- A :: previous ATC
- future ATC

State invariant is kept.

Algorithm - Path-Finding

- Start with a queue with the initial state
- Within each iteration until queue is empty,
  - Pop all current states in the queue
  - Move states reached endpoint to output
  - Step popped states
  - Push newly generated states into the queue

The iteration will end when there's no possible search, or we've searched all potential paths. Finding a valid path won't terminate the iteration.



Formal Specification

We should give some clear definition on the word “correct”, which in formal verification is called “**formal specification**”. Formal specification should be some properties that can be expressed and verified in Coq.

In our context, we regard a path “correct” if:

- The path is a connected path.
- The path follows the ATC command.
- The start of the path is the input startpoint.
- The end of the path is the input endpoint.
- The path is a valid path in the input ***undirected*** graph.

Formal Specification - Example

The property “the path is a valid path in the graph” is expressed in Coq that,

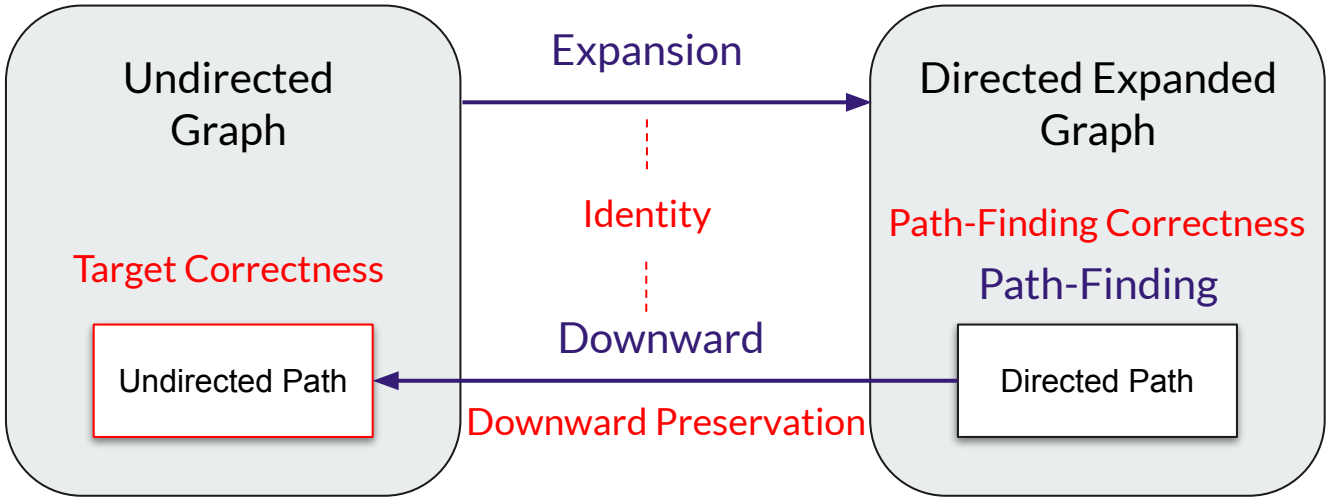
```
Theorem output_path_in_graph :
   forall start_v end_v ATC G output path,
   Some output = find_path start_v end_v ATC G ->
   In path output -> path_in_graph path G.
```

It means “**for all input variables, if a path exists in the output, then the check function** `path_in_graph` **should return true**”, where `path_in_graph` is a check function for a valid path in graph. (A **check function** will return True or False whether some property holds.)

The other properties are defined similarly, we write a function to check whether the property holds, and declare the function always return true on all outputs.

Verification - Overview Logic

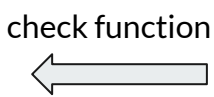
It's hard to directly verify the complete composed algorithm. We divide the overall correctness into three parts corresponding to the three sub-algorithms.



Correctness - Path-finding

The path-finding correctness is encoded into **five formal specifications**. Each formal specification corresponds to one of the formal specifications we described in the overall correctness, but here they're evaluated in the **directed expanded graph**. They are encoded with five check functions and verified in independent theorems.

1. Path is connected.
2. Path starts correct.
3. Path ends correct.
4. Path follows ATC command.
5. Path is in the graph.



`path_conn`

`path_starts_with_vertex`

`path_ends_with_vertex`

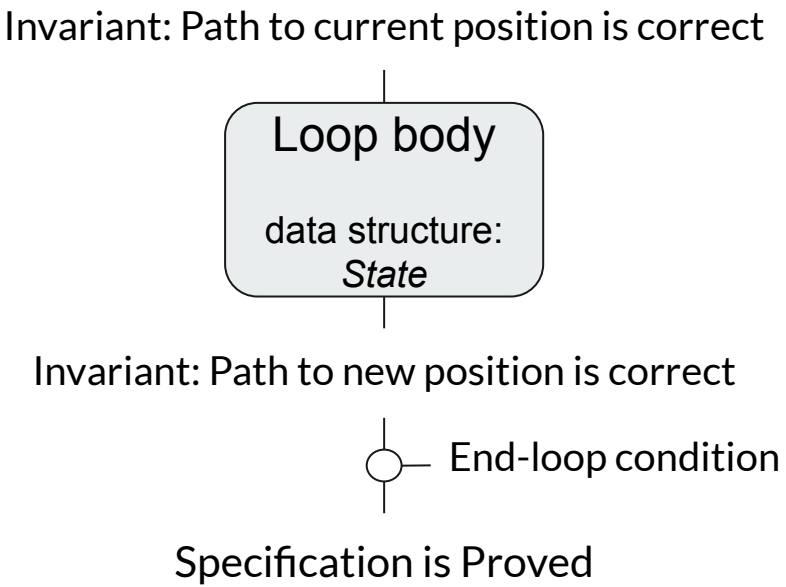
`path_follow_atc`

`path_in_graph`

Verification - Path-finding

The body of the path-finding algorithm is a loop (or recursive call in implementation). So it's natural to prove the specification by invariant. We prove the correctness of path-finding by combining two lemmas:

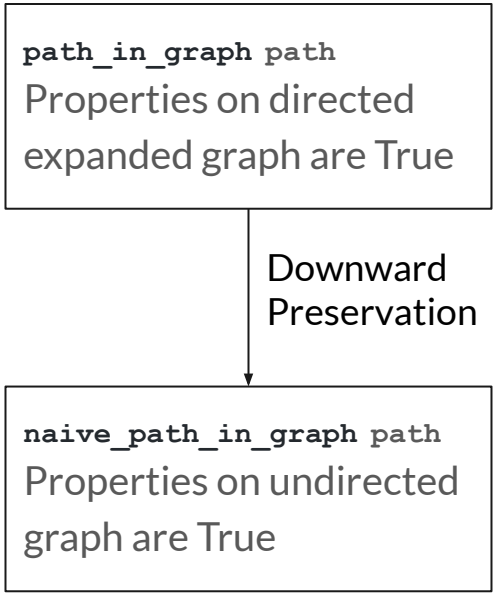
- Invariant is preserved
- The loop ends when the specification is matched



Correctness - Downward

The downward preservation describes that **formal specifications still hold when applying downward to both path and graph**. We similarly write five check functions on undirected graph corresponding to the five check functions on directed graph.

Since the directed expanded graph contains more information than undirected graph, so we can easily prove the downward preservation by filtering unused information.



Example of “in graph” property

Correctness - Identity

We encode identity as:

for all undirected graph `G`,

```
downward (expansion G) = G
```

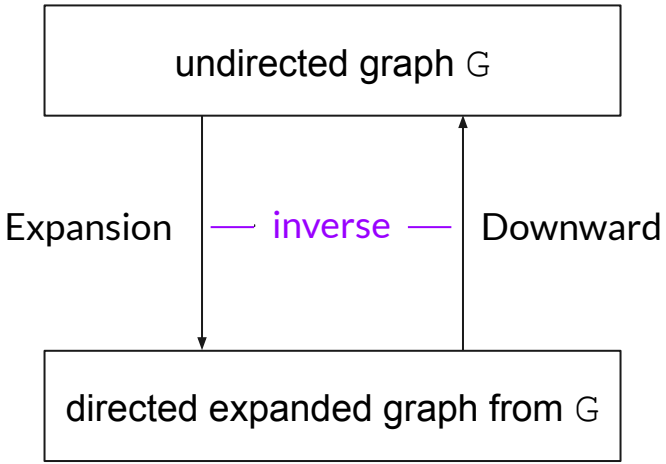
We prove by separating the equality to two statement:

- for all `edge ∈ G`,

  `edge ∈ downward (expansion G)`
- for all `edge ∈ downward (expansion G)`,

  `edge ∈ G`

Then we just need to show the edge is either preserved or transformed by all of the operations in the expansion algorithm and downward algorithm.



Extraction to OCaml

Coq code is not user-friendly to run, so we provide the OCaml code extracted from our implementation. The extraction is provided in Coq standard library, so the extracted code can be regarded formally correct.

We test the result on some large airports, such as DTW. The test confirms that our algorithm can successfully find the path.

```
[
(from input to Ch on taxiway C);
(from Ch to BC on taxiway C);
(from BC to AC on taxiway C);
(from AC to AB on taxiway A);
(from AB to AA3 on taxiway A);
(from AA3 to A3r on taxiway A3);
]
```

A sample output of extracted OCaml code.

Conclusion & Our Effort

In conclusion, we implement an airport path-finding algorithm in Coq, and formally write and verify the formal specifications of the correctness.

Here's some interesting data record of our effort on the project:

- Compile time: about 10 min
- Lines of code: 285 of algorithm, 1983 of proof
- Count of rewrites: 9 big rewrites, countless small rewrites

Thanks for watching!

Siyuan He: hesy@umich.edu
Ke Du: madoka@umich.edu

Limitation

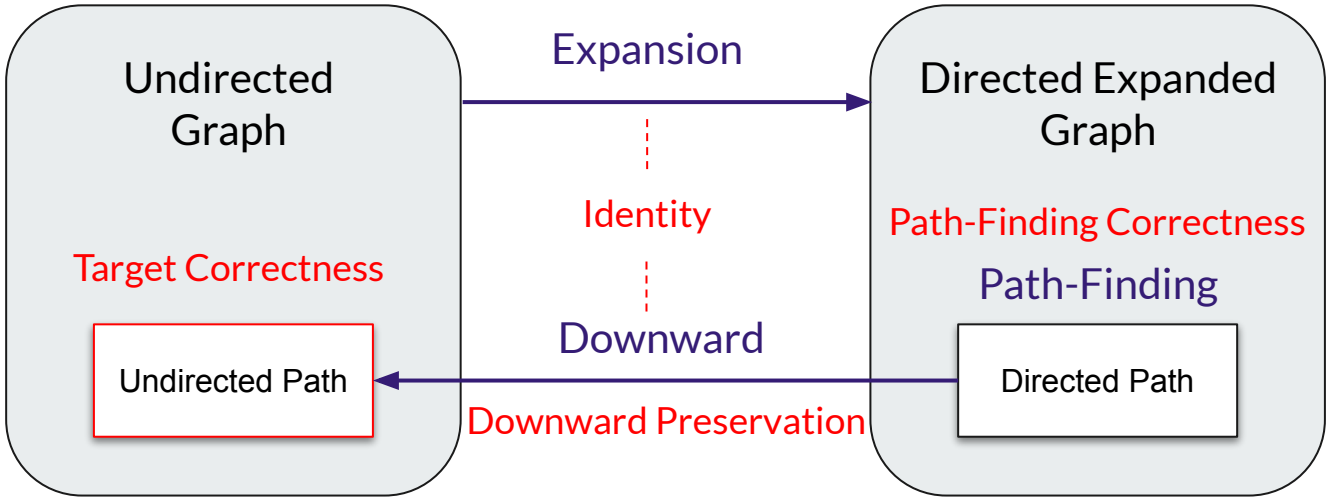
Correctness - Proof Strategy

Since it's hard to directly prove the correctness of the composite algorithm, alternatively we divide the correctness into three sub-theorems:

- Correctness of Path-Finding algorithm on directed expanded graph
- Downward Preservation of correctness
- Identity of Expansion algorithm and Downward algorithm

The correctness is proved in Coq.

Correctness - Proof Logic



Prove the correctness statement of the directed path found on the directed expanded graph.
By proving the state invariant.

Preservation: The correctness can preserve on the downward algorithm.

Identity: The expansion and downward algorithm are correct.

Correctness

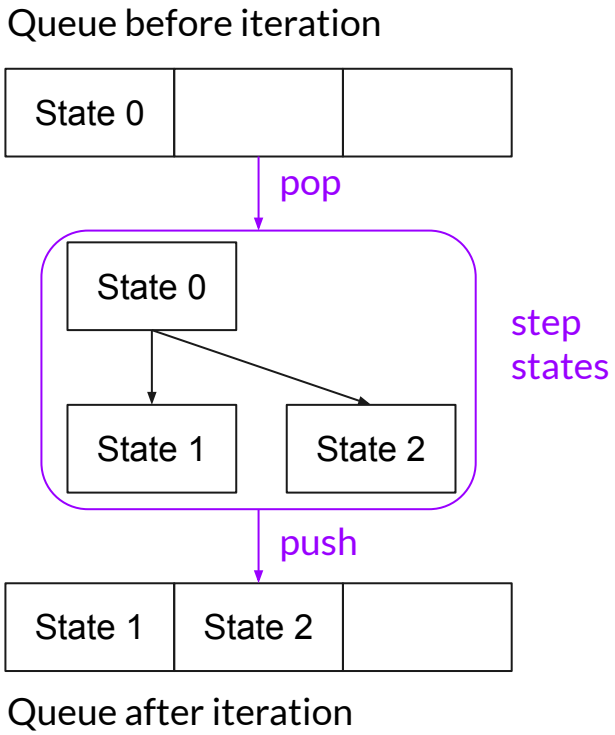
We regard the output path is correct if:

- The path is a connected path.
- The path follows the ATC command.
- The start of the path is the input startpoint.
- The end of the path is the input endpoint.
- The path is a valid path in the ***undirected*** graph.

Algorithm - Path-Finding

- Start with a queue with the initial state
- Within each iteration until queue is empty,
  - Pop all current states in the queue
  - Move states reached endpoint to output
  - Step popped states
  - Push newly generated states into the queue

The algorithm has a bound for maximum iteration. The program will terminate when it reaches the bound or find the endpoint.



Verification - Overview Design

Ideally, we should **directly verify the output of the complete algorithm**. However, it saves much effort to **prove the property of each sub-algorithm**, so we use this strategy in our verification.

We divide the overall correctness into three parts corresponding to the three sub-algorithm:

- Correctness of Path-Finding algorithm on directed expanded graph
- Downward Preservation of correctness
- Identity of Expansion algorithm and Downward algorithm

Formal Specification

∀ graph ATC_command:=(start_vertex, end_vertex, taxiways) path,

IF path ∈ (algorithm ATC_command graph) THEN

(path_interconnected path) AND

(path_in_graph path graph) AND

(path_follow_taxiways path taxiways) AND

(path_first_vertex path start_vertex) AND

(path_last_vertex path end_vertex).

Algorithm

Our algorithm consists of three individual sub-algorithms:

- Expansion Algorithm - from undirected graph to directed expanded graph
- **Path-finding Algorithm** - path-finding on directed expanded graph
- Downward Algorithm - from directed expanded graph to undirected graph