Simple and Fast Strong Cyclic Planning for Fully-Observable Nondeterministic Planning Problems

Jicheng Fu[1], Vincent Ng[2], Farokh Bastani[2], and I-Ling Yen[2]

[1]The University of Central Oklahoma
[2]The University of Texas at Dallas

Goal

- To solve strong cyclic planning problems from a Fully-Observable Nondeterministic planning domain

Goal

- To solve strong cyclic planning problems from a Fully-Observable Nondeterministic planning domain

Goal

- To solve strong cyclic planning problems from a Fully-Observable Nondeterministic planning domain

- A planning problem is a triple $\langle s_0, g, \Sigma \rangle$, where
  - s_0 is the initial state,
  - g is the goal condition, and
  - Σ is the planning domain

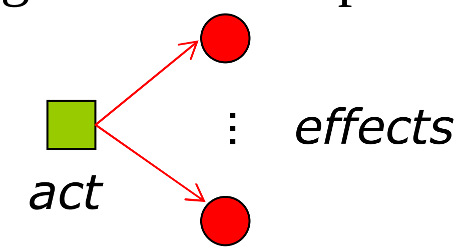
Goal

- To solve strong cyclic planning problems from a Fully-Observable Nondeterministic planning domain

Goal

- To solve strong cyclic planning problems from a Fully-Observable Nondeterministic planning domain
- Informally, in a nondeterministic planning domain,

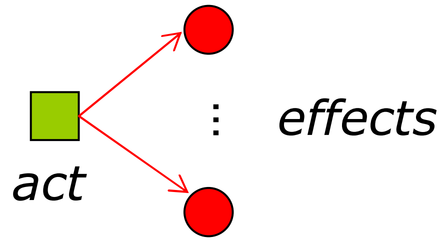
    an action may generate multiple effects



Goal

- To solve strong cyclic planning problems from a Fully-Observable Nondeterministic planning domain
- Informally, in a nondeterministic planning domain,

  an action may generate multiple effects



- Formally, a nondeterministic domain

  is a 4-tuple $\Sigma = (P, S, A, \gamma)$
  - P is a finite set of propositions;
  - $S \subseteq 2^P$ is a finite set of states in the system;
  - A is a finite set of actions; and
  - $\gamma: S \times A \rightarrow 2^S$ is the state-transition function

Goal

- To solve strong cyclic planning problems from a Fully-Observable Nondeterministic planning domain

Goal

- To solve strong cyclic planning problems from a Fully-Observable Nondeterministic planning domain

- Full observability

  The states of the world are fully observable

Goal

- To solve strong cyclic planning problems from a Fully-Observable Nondeterministic planning domain

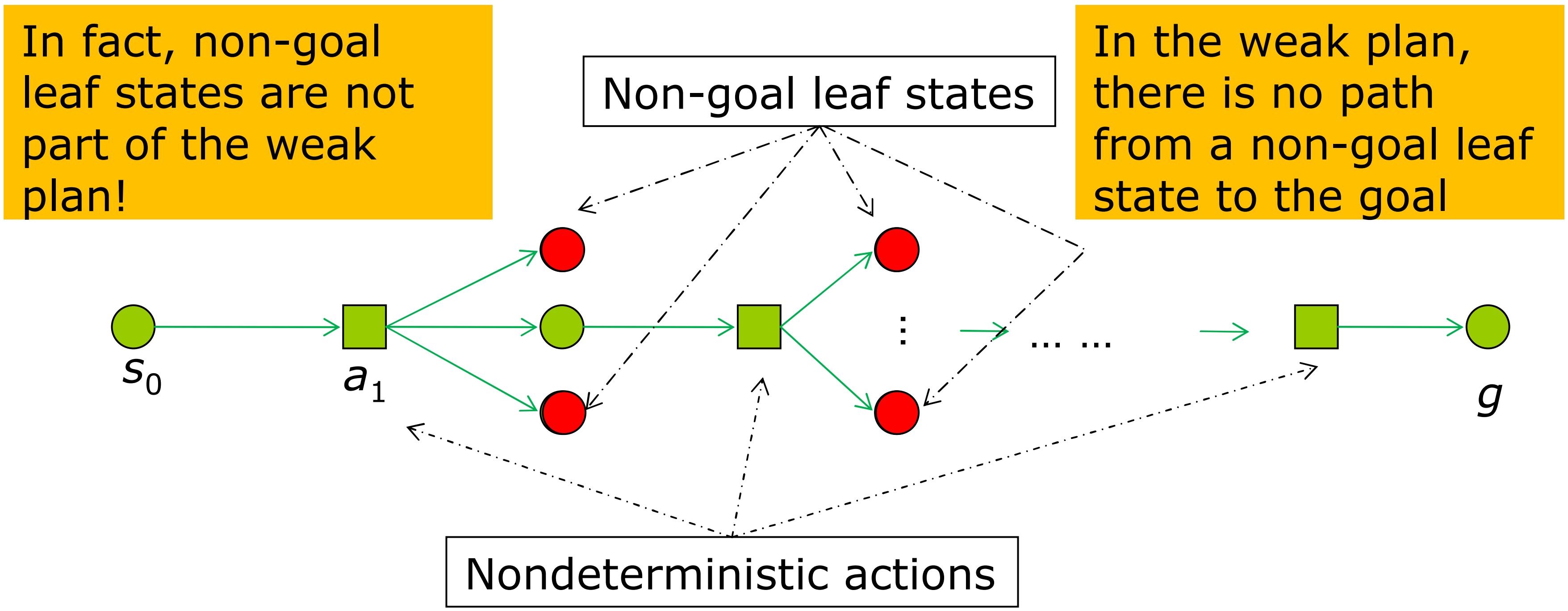
Goal

- To solve strong cyclic planning problems from a Fully-Observable Nondeterministic planning domain

- Strong cyclic planning
  - refers to a particular type of solutions to nondeterministic problems
  - different from so-called weak planning

Weak Planning Solutions

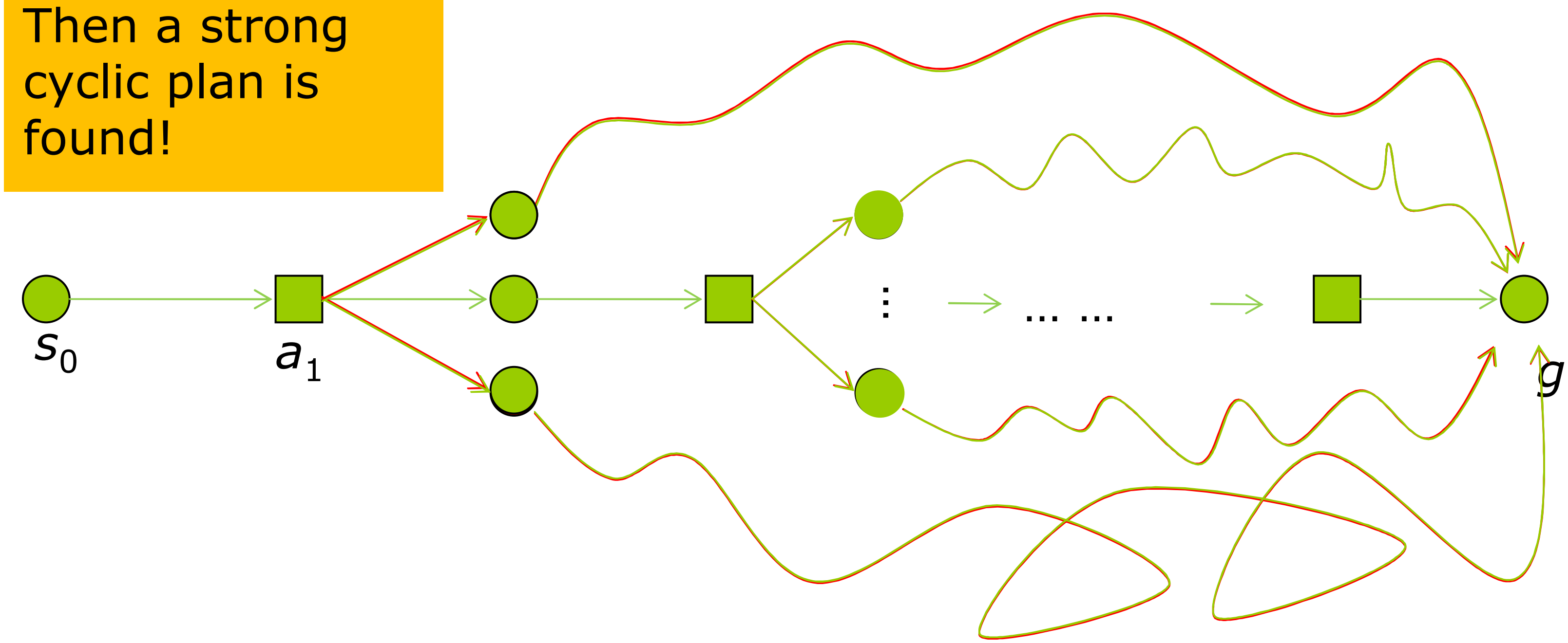
- Solutions where there is a chance to achieve the goal



Strong Cyclic Planning Solutions

- Prescribe actions for all possible non-goal leaf states
  - Find a path for each non-goal leaf state to the goal state
  - May loop indefinitely
  - But contain no dead-ends
  - More difficult than finding weak planning solutions

Then a strong cyclic plan is found!



Representing a Plan

- Regardless of whether a plan is weak or strong cyclic, we can represent it as a policy π
  - a partial function mapping states to actions

- More formally, policy $\pi : S_\pi \rightarrow A$
  - consists of state action pairs (s, a) such that $\pi(s) = a$
  - defines which action to take under state s

How to Generate a Strong Cyclic Plan

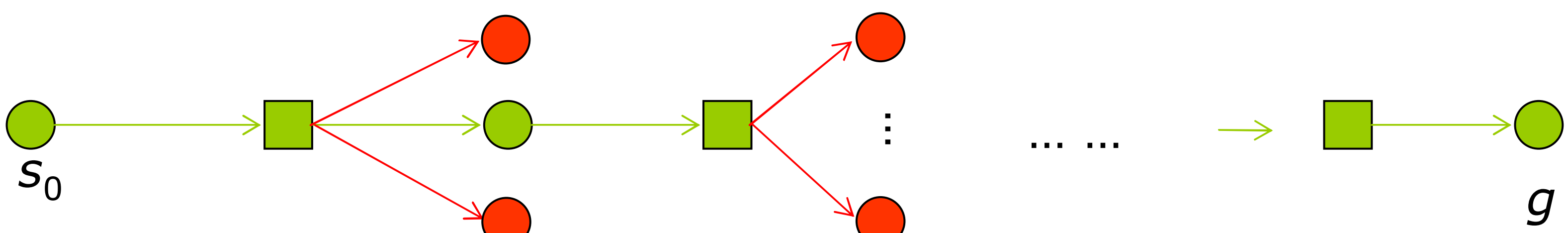
- Given a planning problem with initial state s_0 and goal state g, we employ a 3-step Basic algorithm
  - motivated by work in **Incremental contingency planning** [Dearden *et al*, 2003], **FF-replan** [Yoon, Fern, and Givan, 2007], and **NDP** [Kuter *et al*., 2008]

Basic Algorithm: Step 1

- Find a **path** (i.e., weak plan) from s_0 to g
  using a classic planner

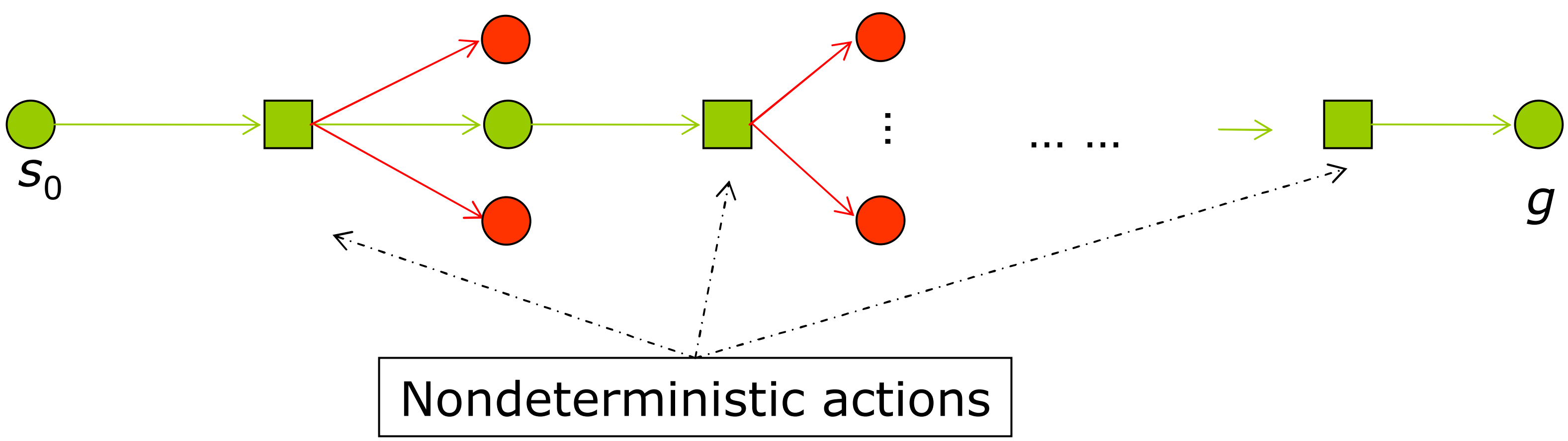
Basic Algorithm: Step 1

- Find a **path** (i.e., weak plan) from s_0 to g
  using a classic planner



Basic Algorithm: Step 1

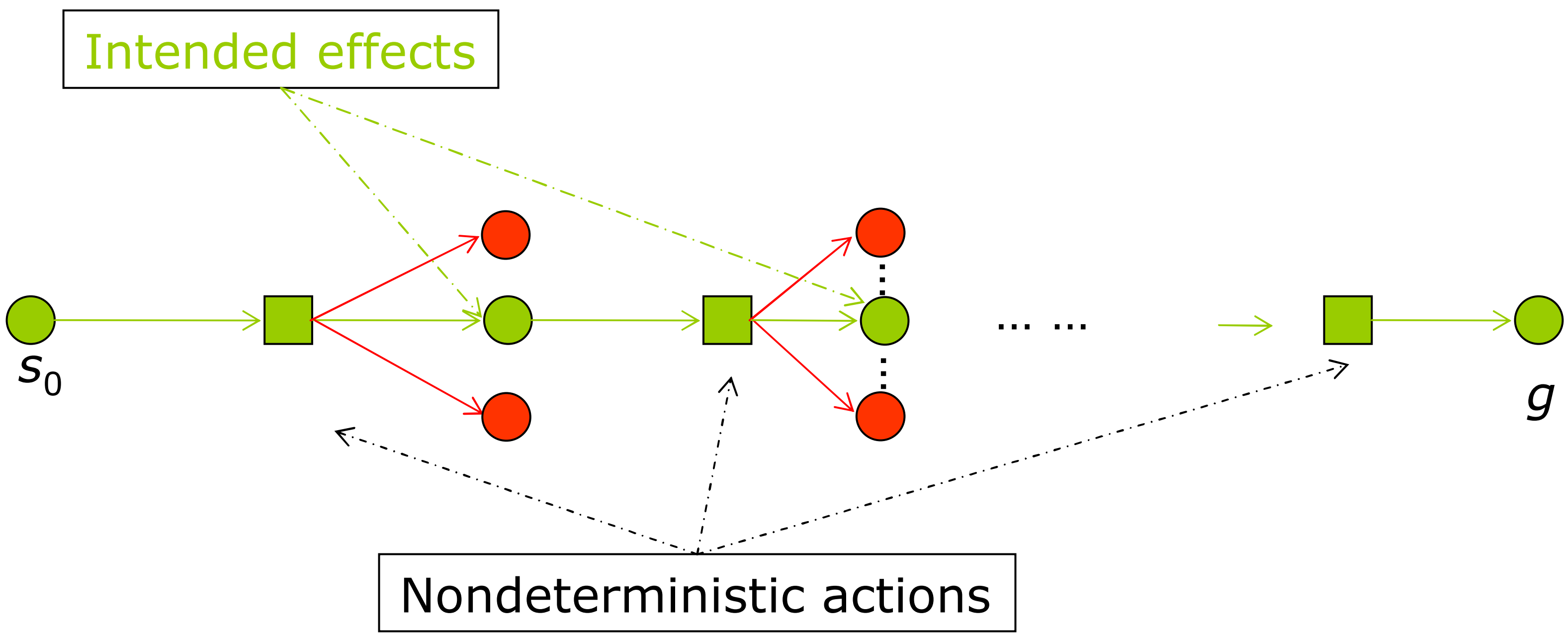
- Find a **path** (i.e., weak plan) from s_0 to g
  using a classic planner



Basic Algorithm: Step 1

- Find a **path** (i.e., weak plan) from s_0 to g
  using a classic planner

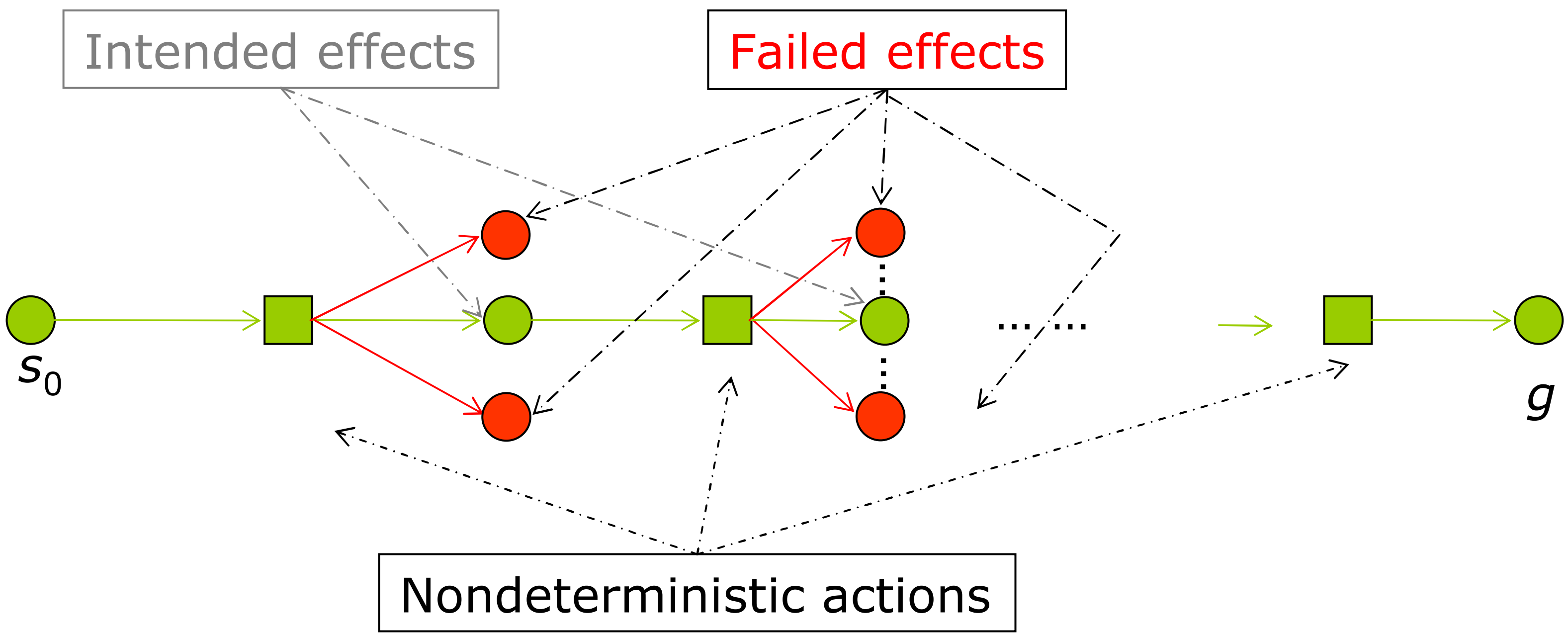
Given a nondeterministic action,
the effect included in the weak plan is its **Intended Effect**



Basic Algorithm: Step 1

- Find a **path** (i.e., weak plan) from s_0 to g
  using a classic planner

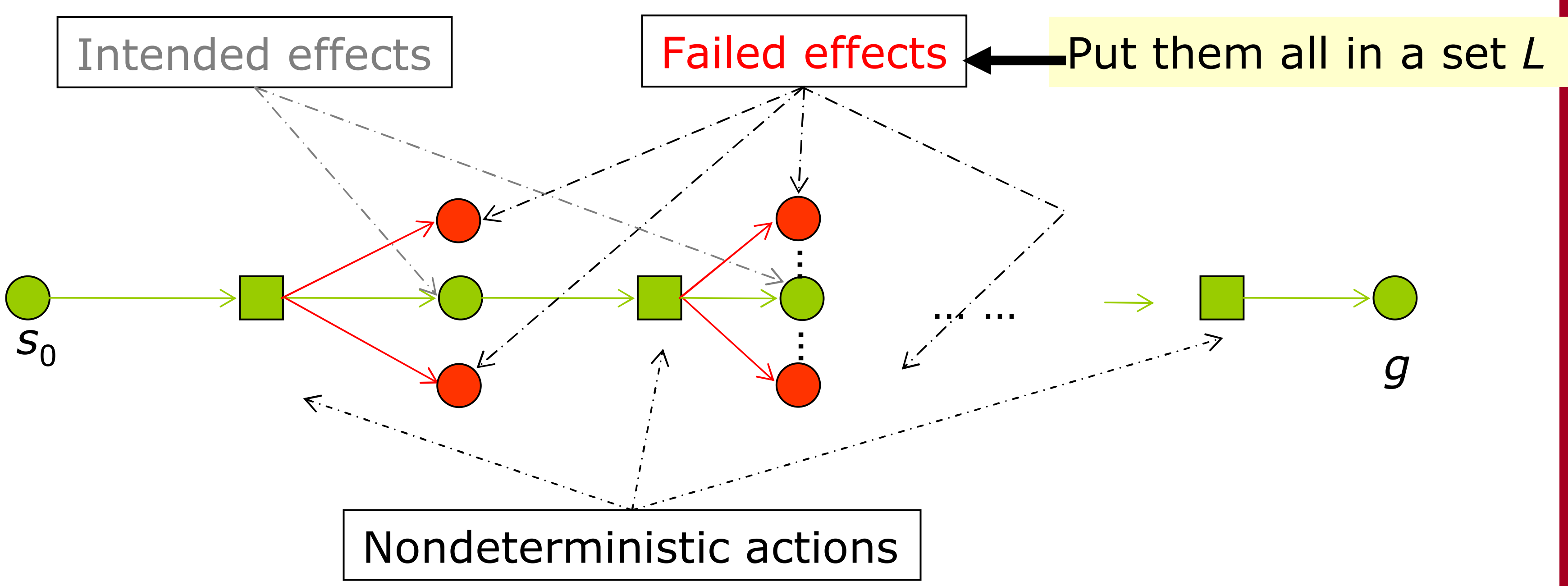
Given a nondeterministic action,
the effect included in the weak plan is its **Intended Effect**
the effects not included in the weak plan are its **Failed Effects**



Basic Algorithm: Step 1

- Find a **path** (i.e., weak plan) from s_0 to g
  using a classic planner

Given a nondeterministic action,
the effect included in the weak plan is its **Intended Effect**
the effects not included in the weak plan are its **Failed Effects**



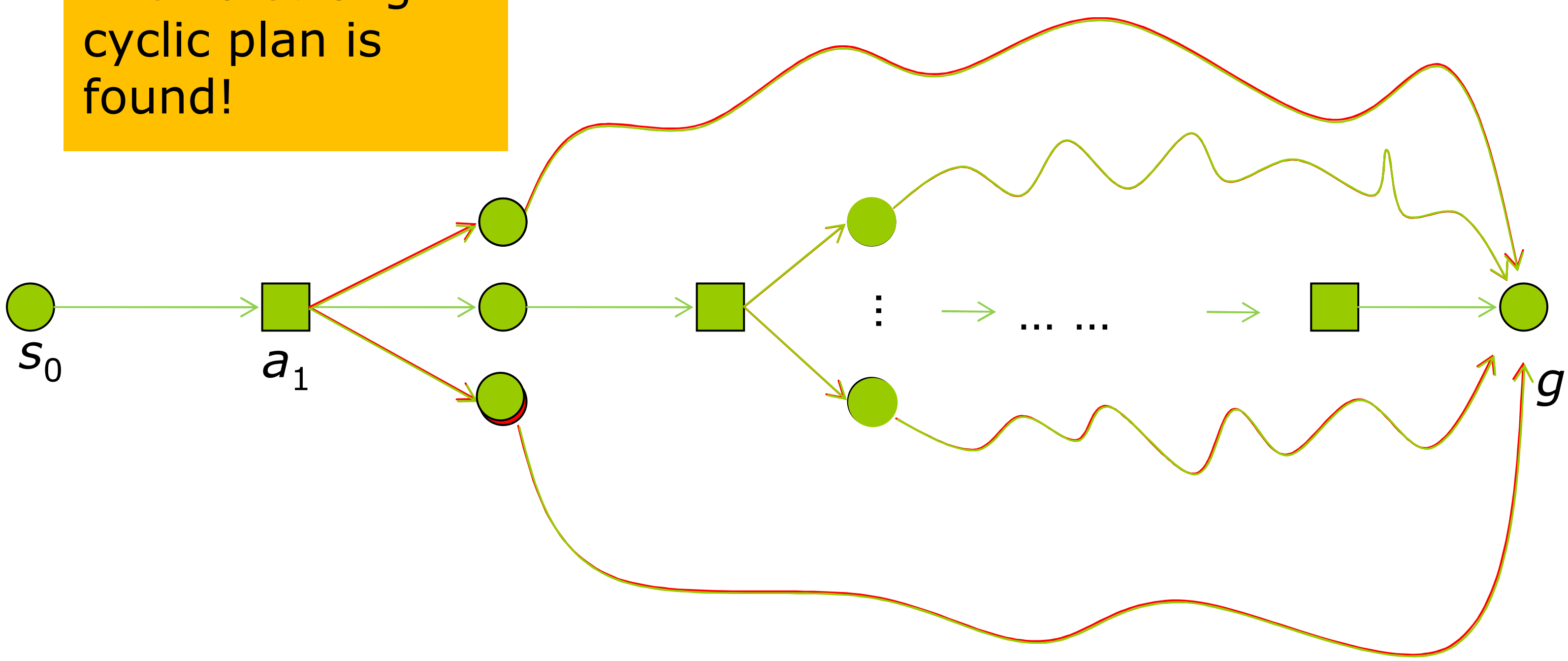
Basic Algorithm: Step 2

- For every failed effect *e*, find a path from *e* to *g*.

Basic Algorithm: Step 2

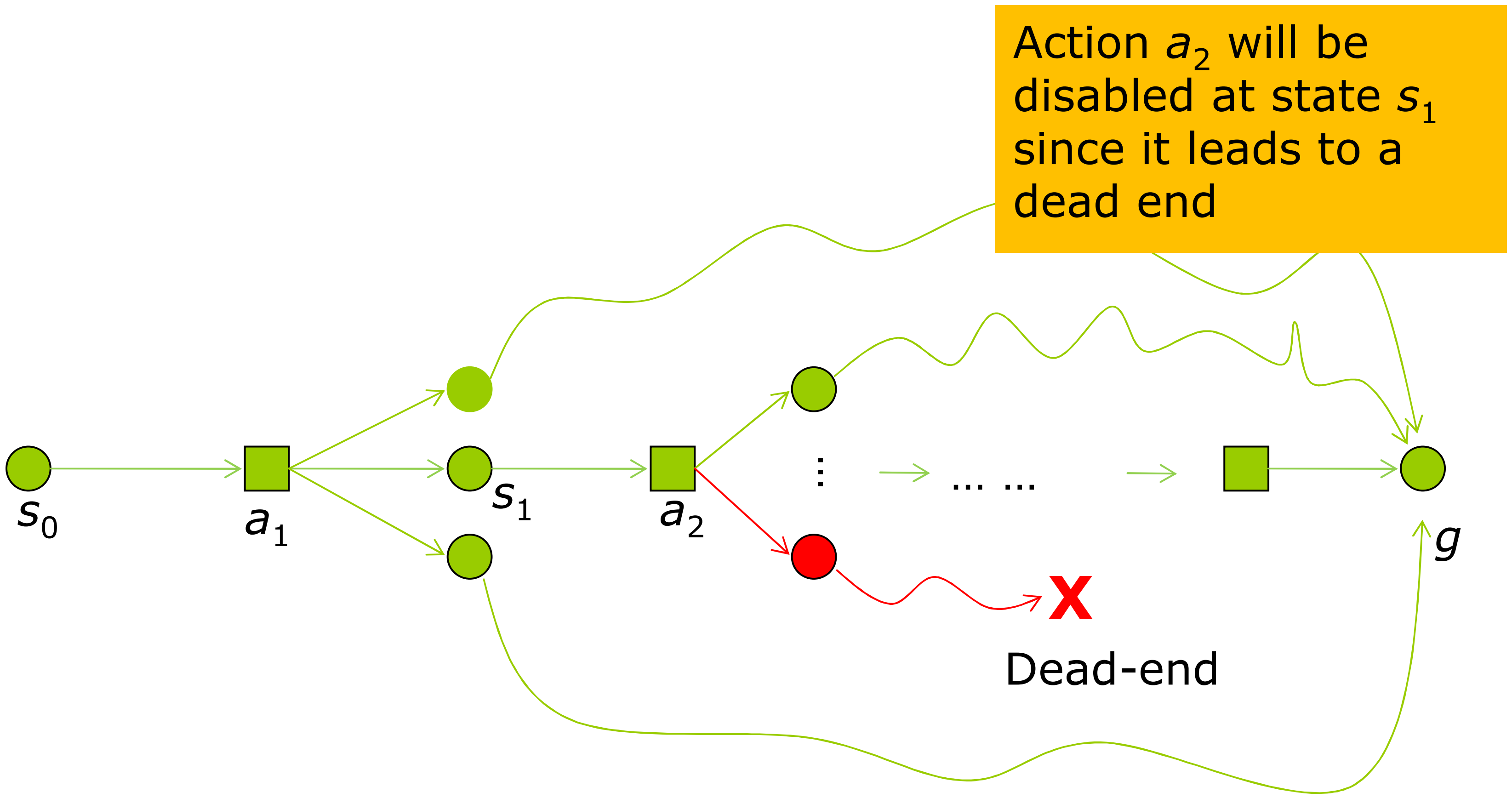
- For every failed effect *e*, find a path from *e* to *g*.

Then a strong cyclic plan is found!



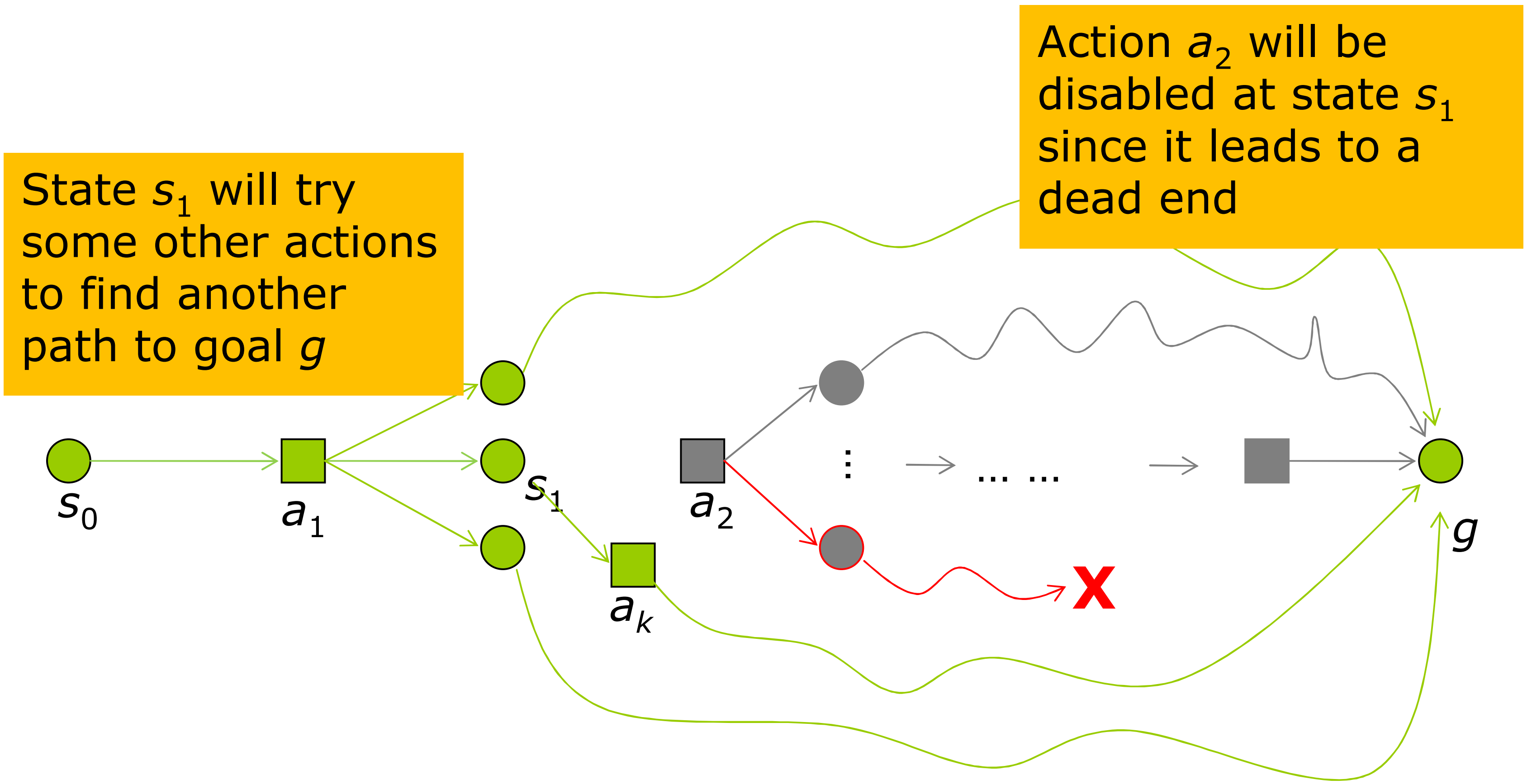
Basic Algorithm: Step 3

- But sometimes we may encounter a dead end. In this case, we need to **backtrack**



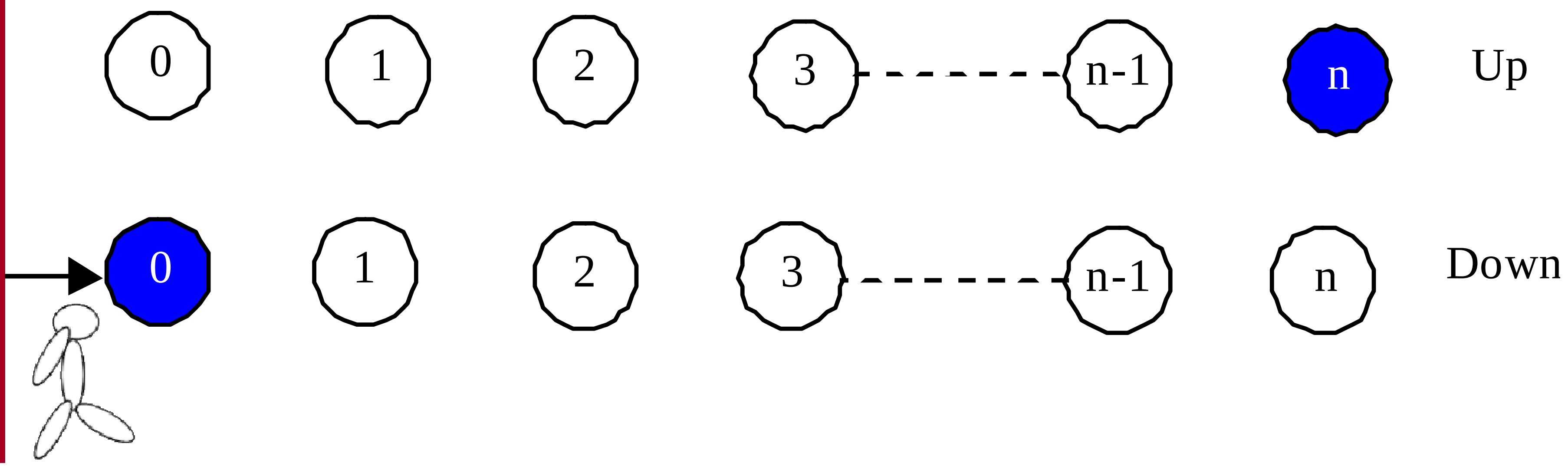
Basic Algorithm: Step 3

- But sometimes we may encounter a dead end. In this case, we need to **backtrack**



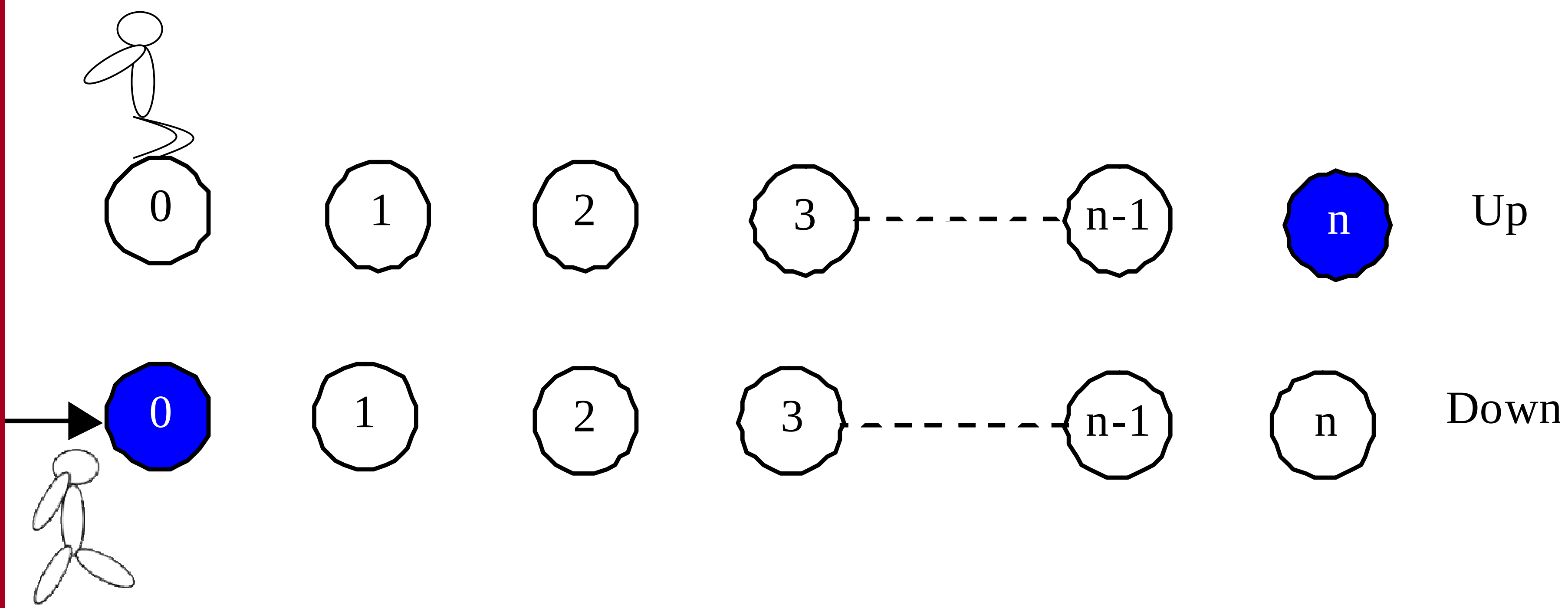
Basic Algorithm: Application to Beam Domain

- A set of **positions** in one of two levels: ***up*** or ***down***
- **Goal:** have the agent move from ***down$_0$*** to ***up$_n$***
- **Three** possible actions can be used to move around



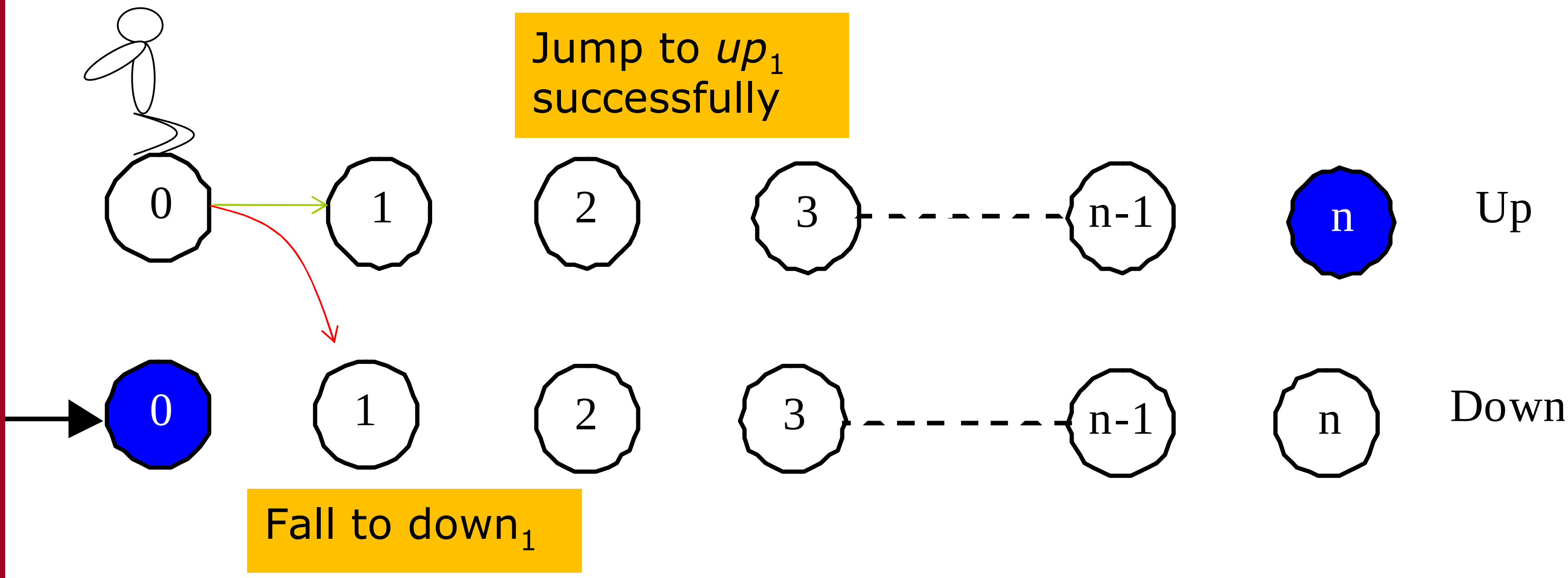
Action 1: Climb

- Climb is deterministic
  - it moves the agent from $down_0$ to up_0.
  - can only be applied to $down_0$



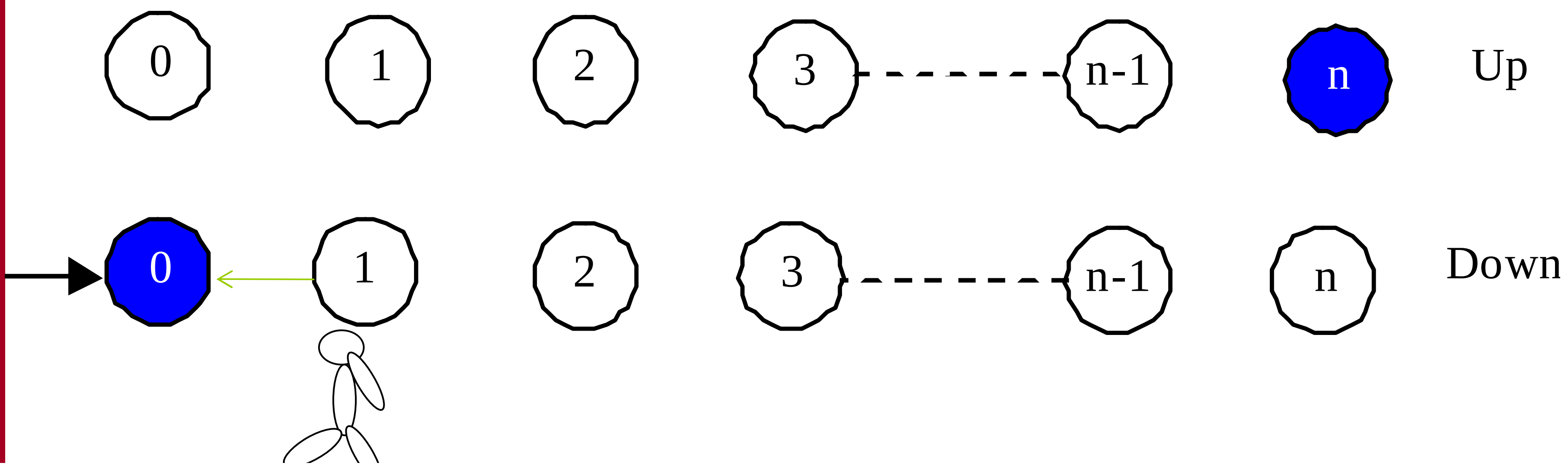
Action 2: Jump

- Jump is nondeterministic
  - can be applied only to a position in the upper level
  - if successful, agent moves to up position to its right
  - if unsuccessful, agent moves to down position to its right



Action 3: Moveback

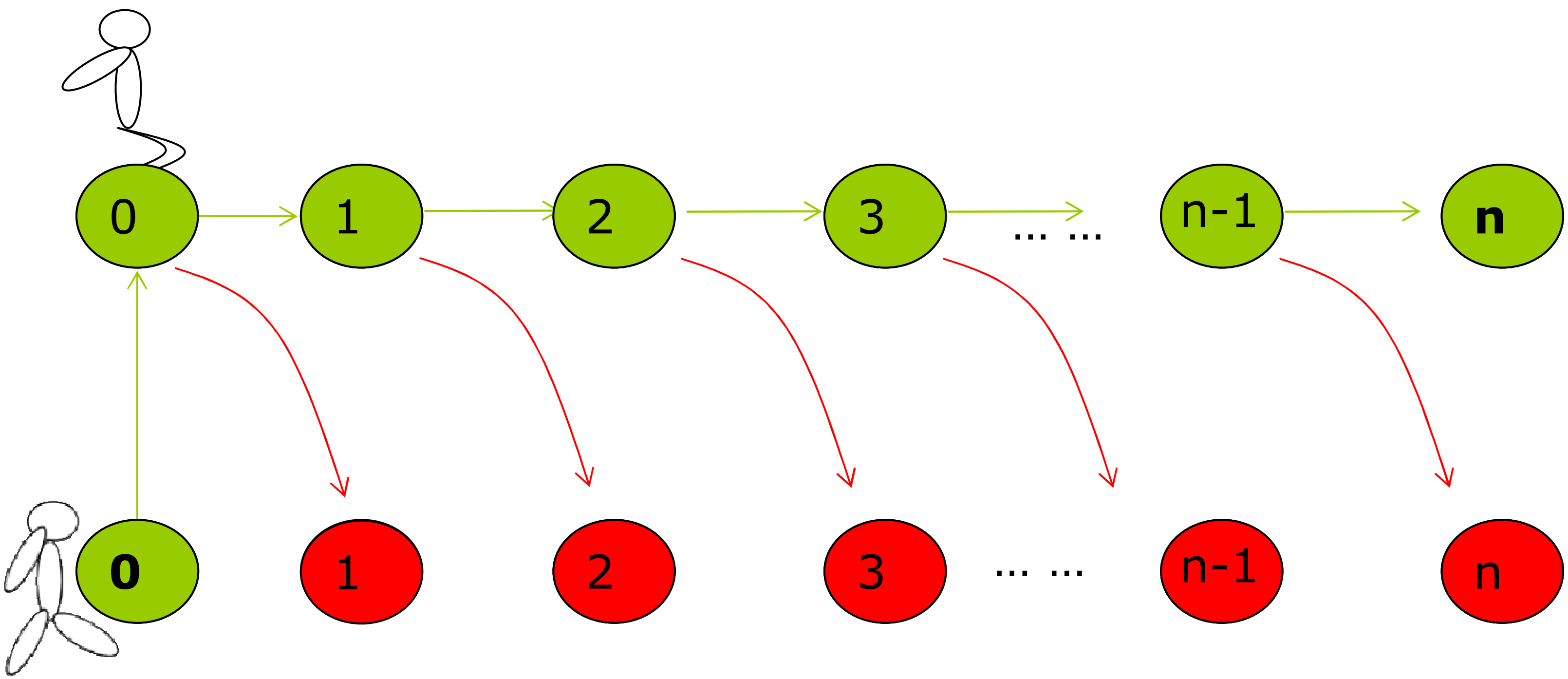
- Moveback is deterministic
  - can be applied only to a position in the lower level
  - moves agent one step to the left



Applying the Basic Strong Cyclic Algorithm

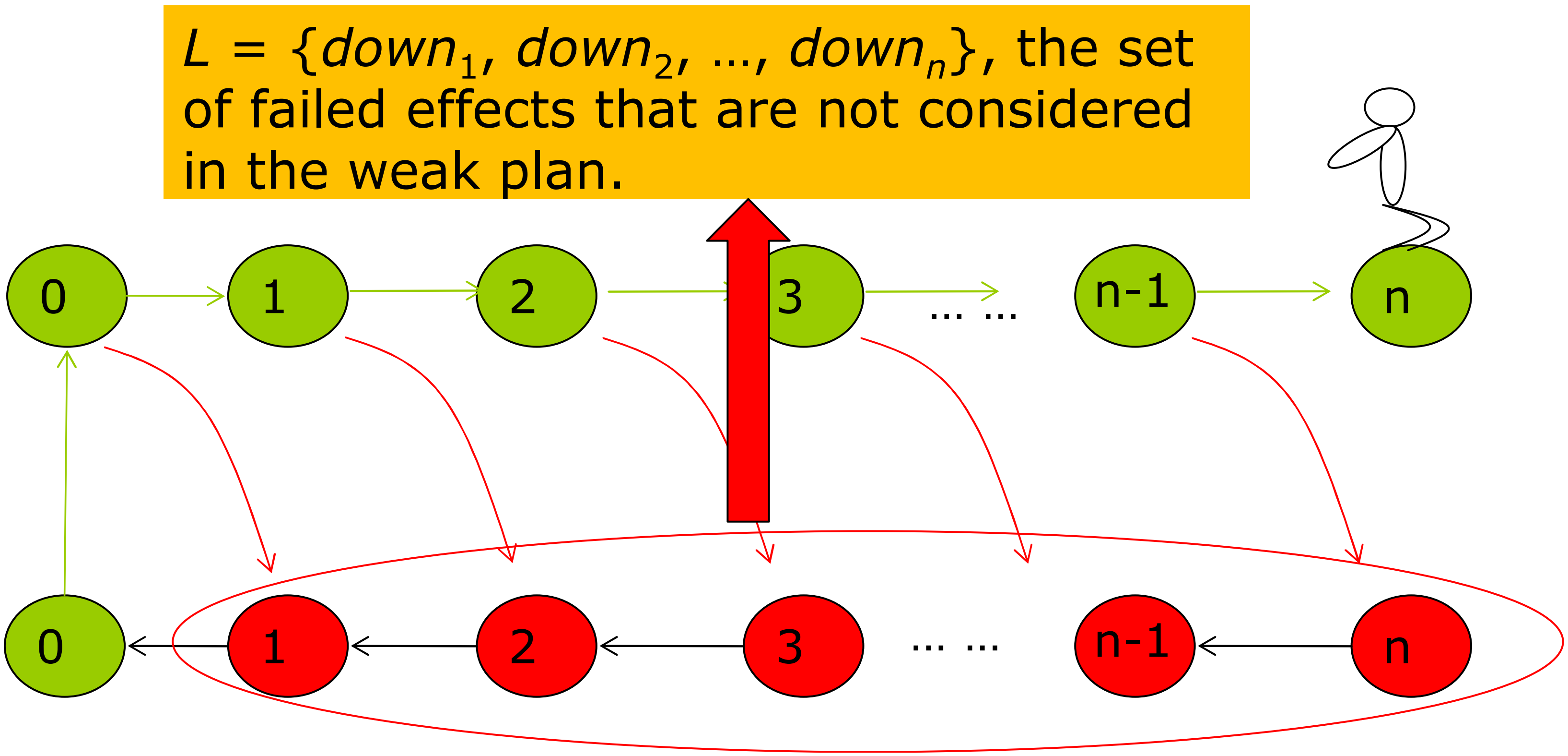
- Initially, $L = \{\text{down}_0\}$
- **Step 1**: Find a path from down_0 to up_n

This is a weak plan because it does not consider the possibility of falling!



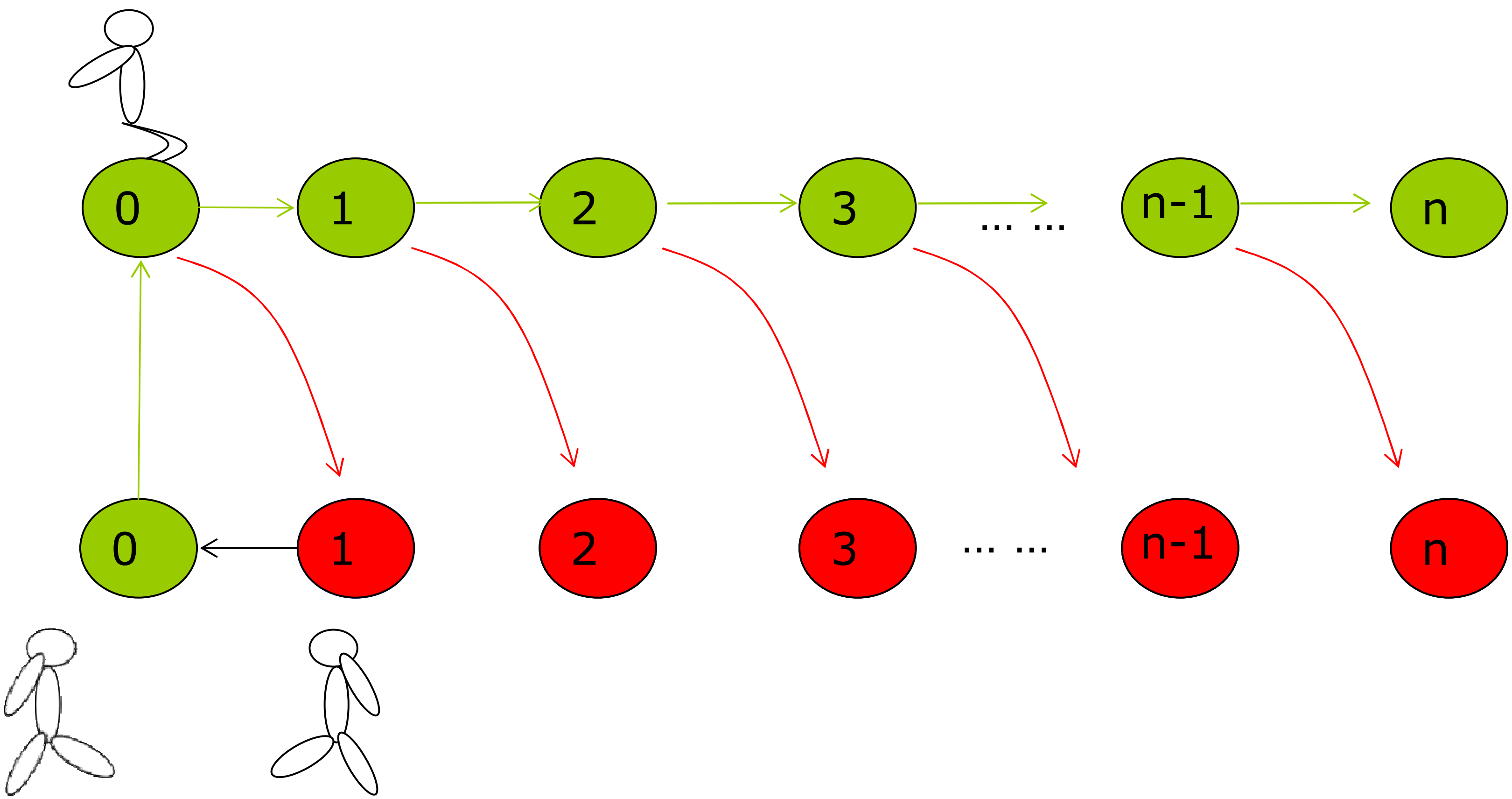
Applying the Basic Strong Cyclic Algorithm

- Initially, L = {down$_0$}
- **Step 1**: Find a path from down$_0$ to up$_n$



Applying the Basic Strong Cyclic Algorithm

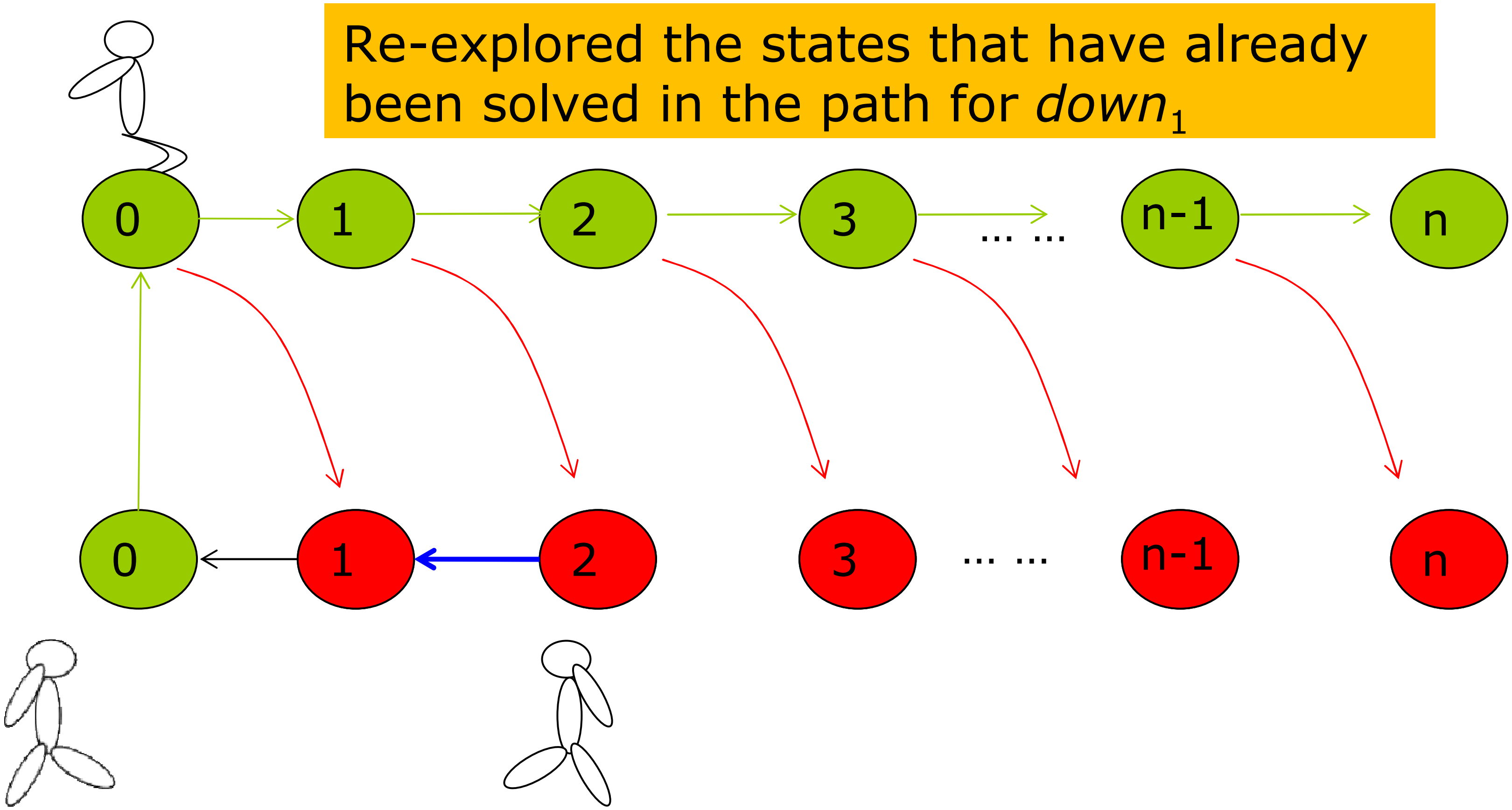
- $L = \{down_1, down_2, \ldots, down_n\}$
- **Step 2**: Find a path from **each** position in L to up_n.
- E.g., for $\boldsymbol{down_1}$, the path to $\boldsymbol{up_n}$ is:



Applying the Basic Strong Cyclic Algorithm

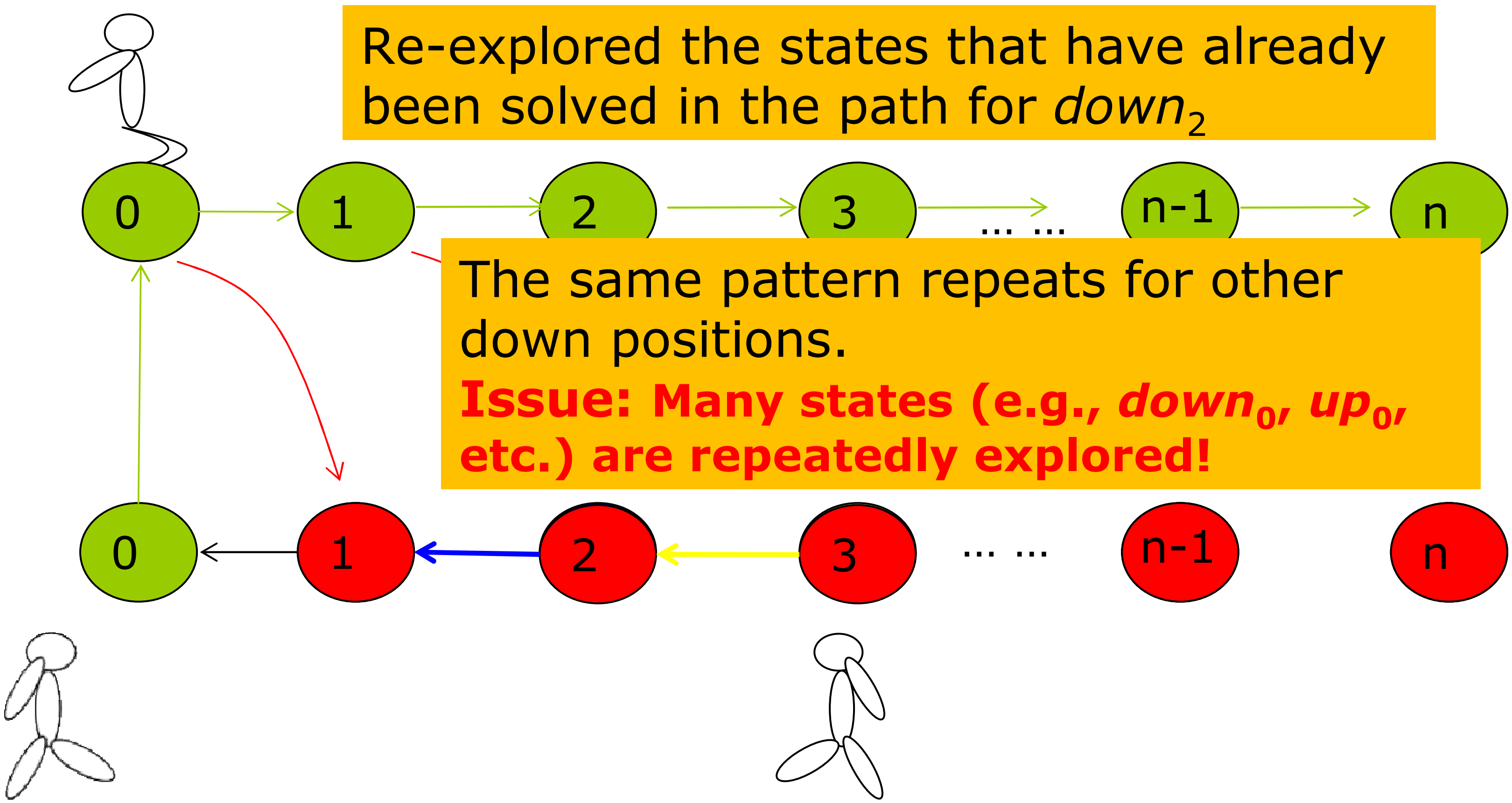
- $L = \{down_1, down_2, \ldots, down_n\}$
- **Step 2**: Find a path from **each** position in L to up_n.
- E.g., for $\boldsymbol{down_2}$, the path to $\boldsymbol{up_n}$ is:

Re-explored the states that have already been solved in the path for *down*$_1$



Applying the Basic Strong Cyclic Algorithm

- $L = \{down_1, down_2, \ldots, down_n\}$
- **Step 2**: Find a path from **each** position in L to up_n.
- E.g., for ***$down_3$***, the path to ***up_n*** is:



So …

- The basic algorithm can be **inefficient**
  - many states can be repeatedly explored

- Goal
  - improve the Basic algorithm w.r.t. planning efficiency and plan size by proposing two extensions

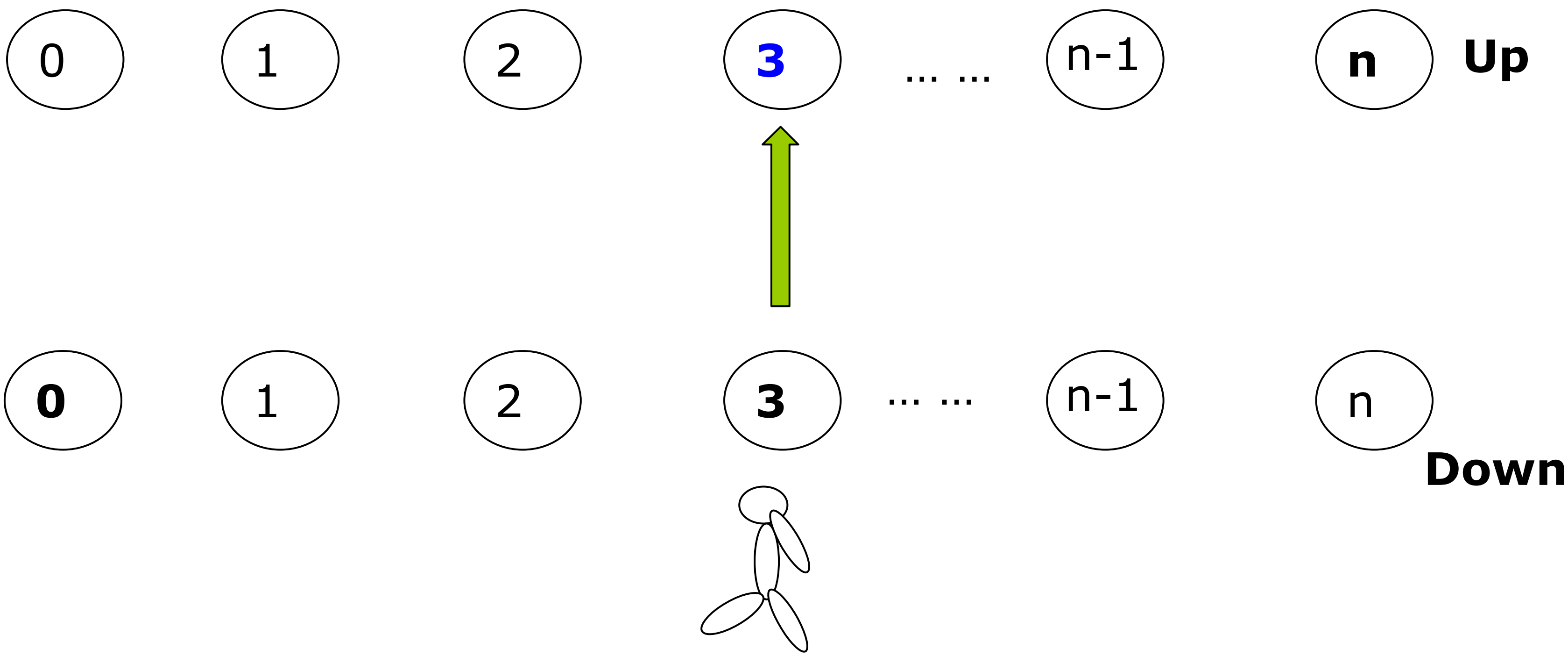
Extension 1: Goal Alternative

- Observation
  - The Basic algorithm attempts to find a path from each failed effect to goal state *g*, which can be far away
- With goal alternative, we attempt to find a path from each failed effect to an alternative goal
  - an alternative goal is presumably closer to the associated failed effect than the overall goal *g*
    - could improve planning efficiency and reduce plan size
  - each failed effect has its own alternative goal
    - i.e., an alternative goal is associated with a failed effect
    - we use the corresponding intended effect as alternative goal

Intended Effect as Alternative Goal

- For each failed effect $down_i$

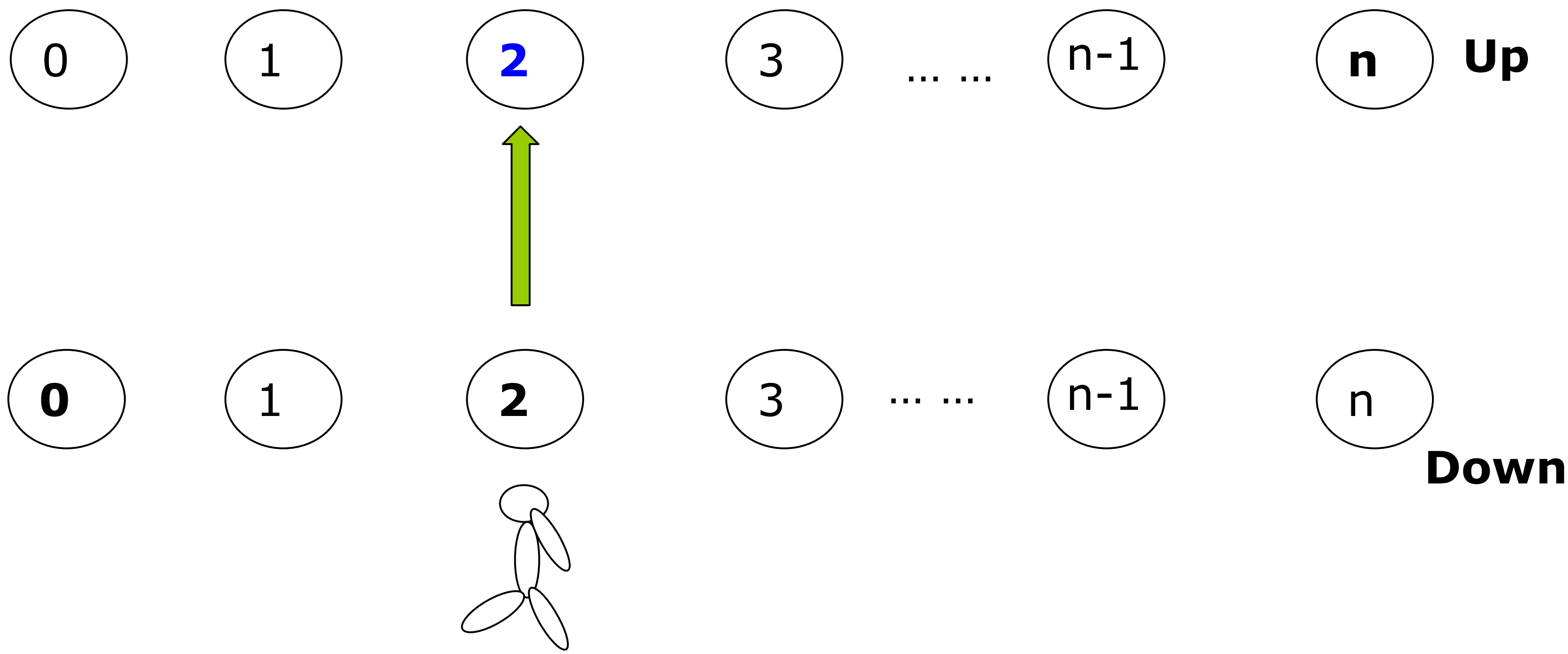
  instead of using up_n as the search goal, we use the intended effect up_i of action Jump(up_{i-1}, up_i) as the search goal



Intended Effect as Alternative Goal

- For each failed effect $down_i$

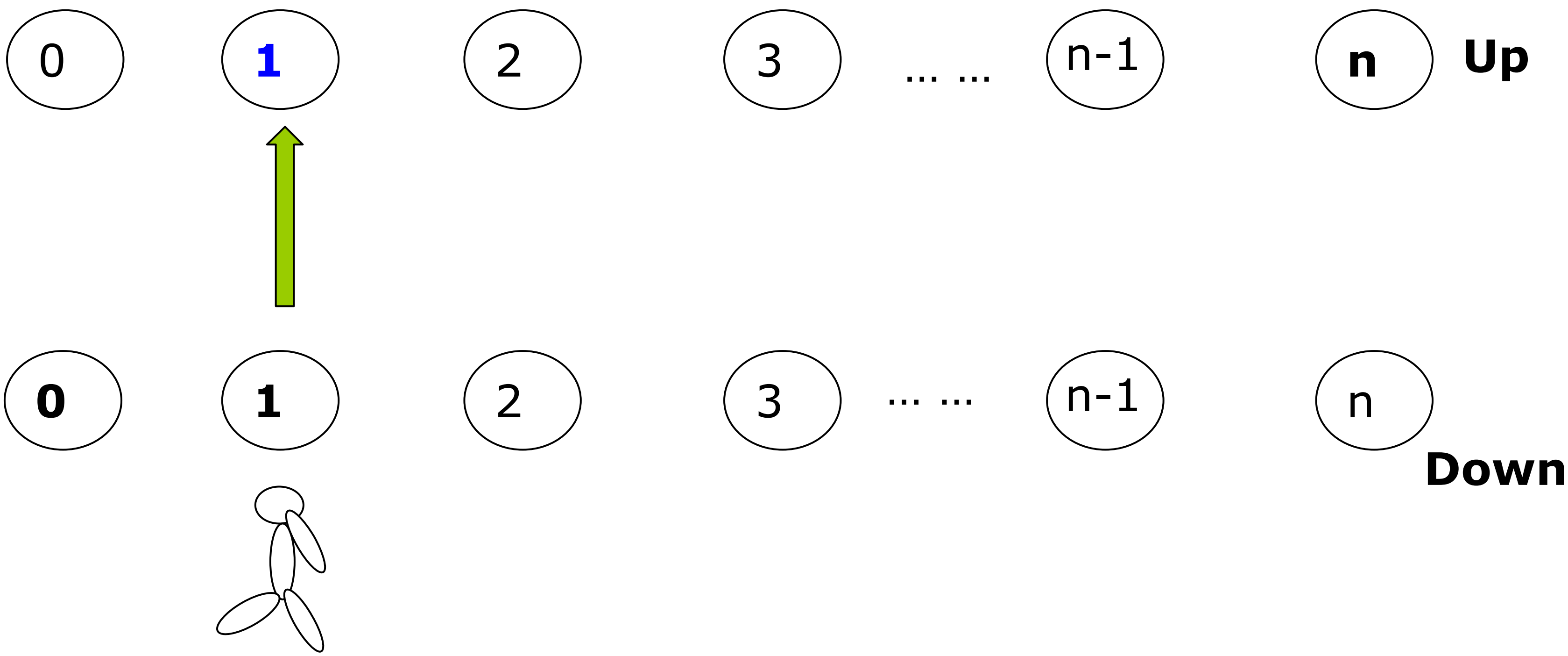
  instead of using up_n as the search goal, we use the intended effect up_i of action Jump(up_{i-1}, up_i) as the search goal



Intended Effect as Alternative Goal

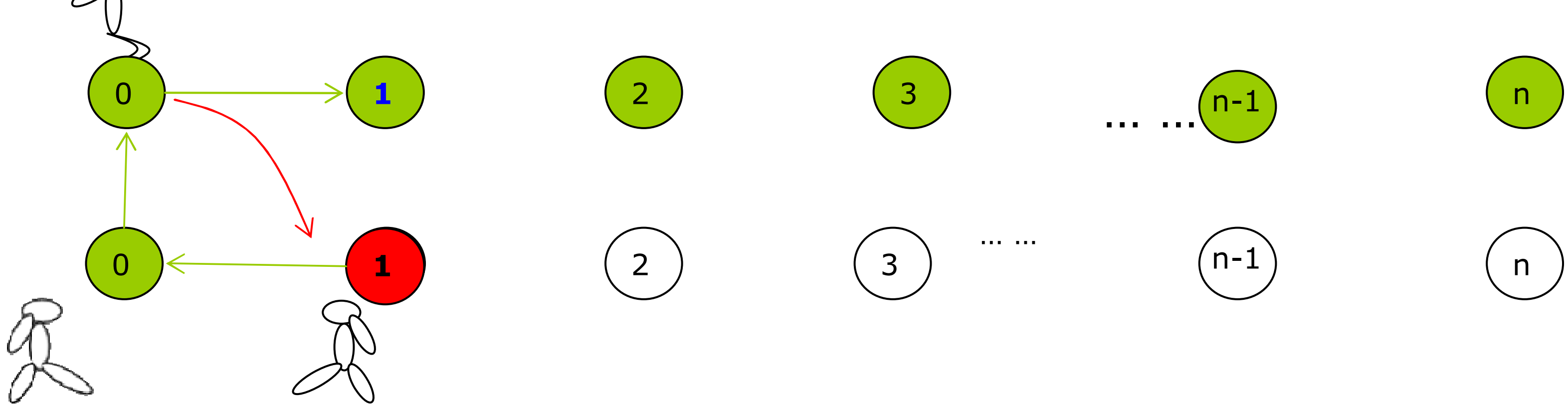
- For each failed effect $down_i$

  instead of using up_n as the search goal, we use the intended effect up_i of action Jump(up_{i-1}, up_i) as the search goal

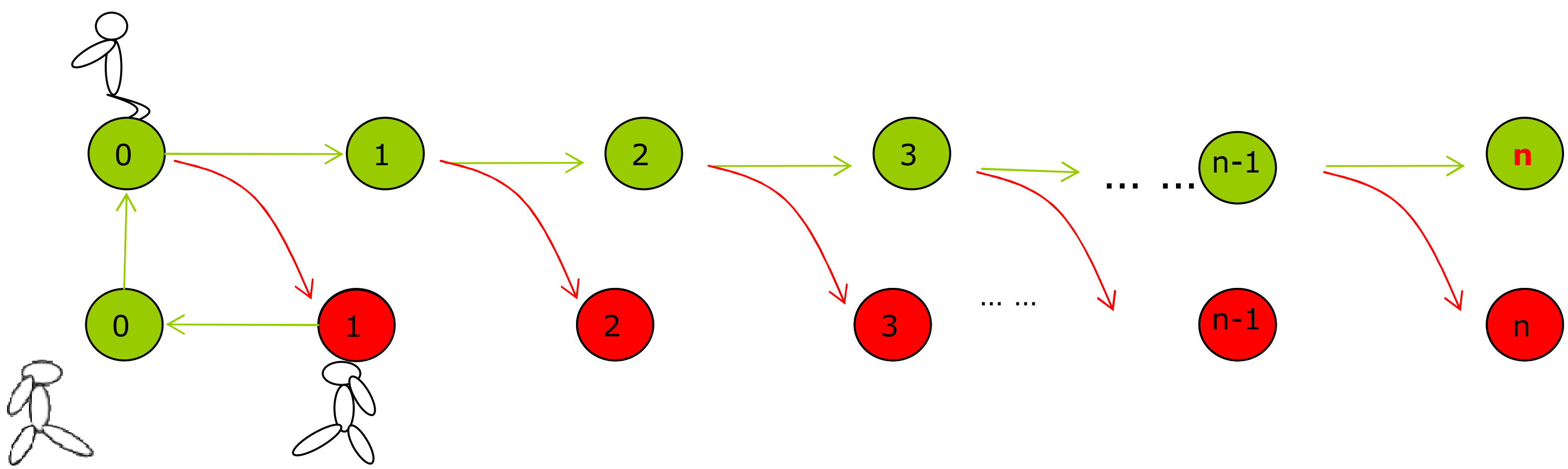


Intended Effect as Alternative Goal

- For failed effect *down*$_1$,
  - If we use goal alternative, the intended effect *up*$_1$ is the search goal



  - If we use the **ultimate goal *up*$_n$** as the search goal
    - The generated weak plan is lengthy



Goal Alternative: A Caveat

- It is possible that a path cannot be established between a failed effect and its alternative goal

- If this happens, we resort to establishing a path from the failed effect to the original goal *g*

Why is Goal Alternative correct?

- By definition, an intended effect $\hat{s}$ is included in some path *wp* to goal *g*, while a failed effect *s* is ignored in *wp*

- Since we have already found a path from $\hat{s}$ to *g*, if we can find a path from *s* to $\hat{s}$, then the path from *s* to $\hat{s}$ can be the solution to $\langle s, g, \Sigma \rangle$

- Hence, much effort is saved by avoiding the search from $\hat{s}$ to *g*.

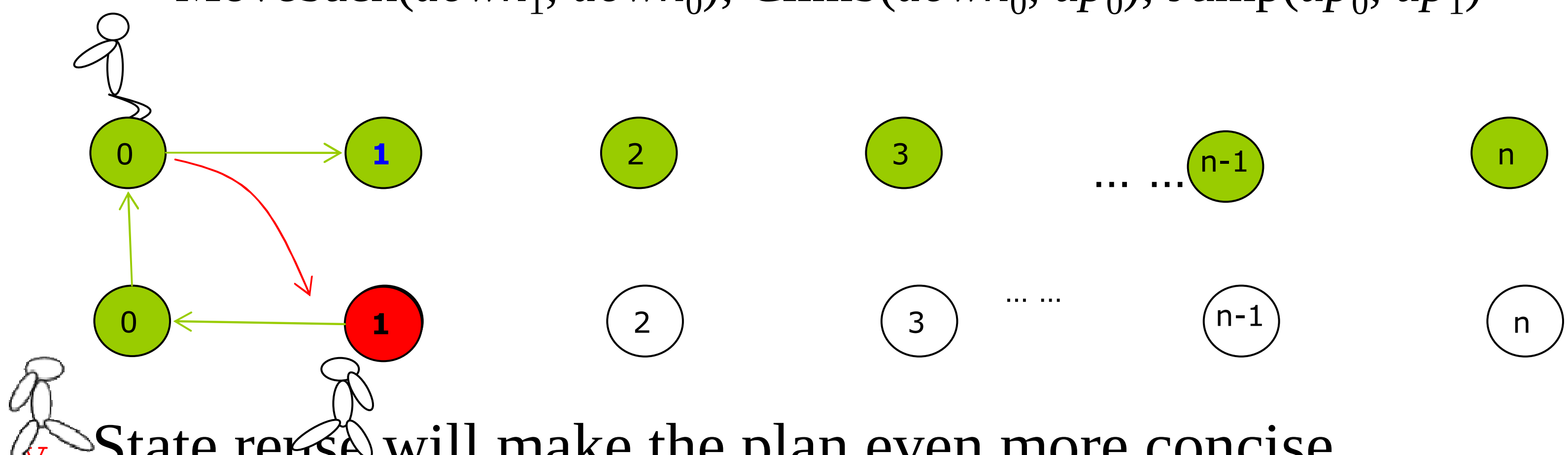
Extension 2: State Reuse

- Observation
  - Even if a state is solved (i.e., a path has been found from *s* to the goal *g*), the Basic algorithm still attempts to solve it every time it is encountered
- State reuse aims to improve planning efficiency by not re-solving a state
  - When searching for a weak plan, if a solved state is encountered, the search stops

State Reuse: Example

- For failed effect $down_1$, goal alternative generates the plan

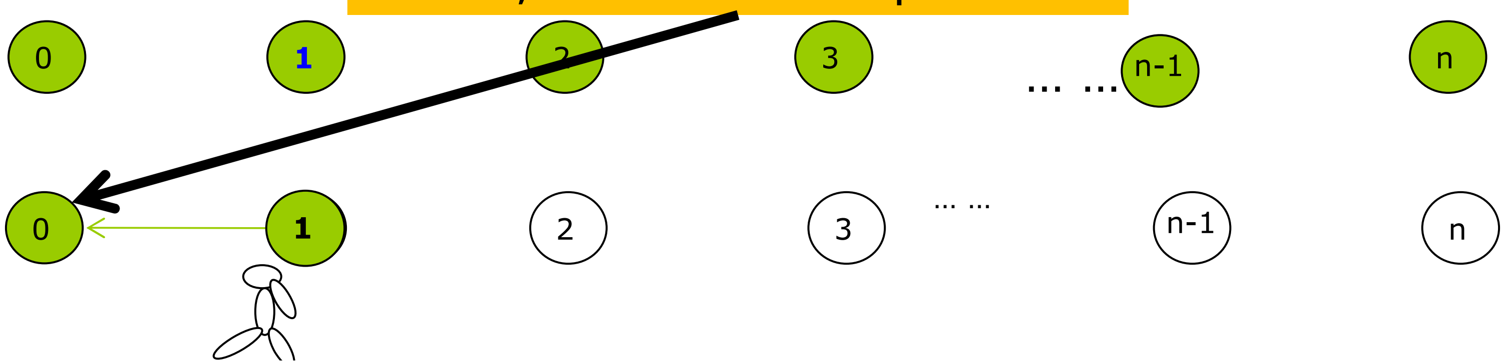
  Moveback($down_1$, $down_0$); Climb($down_0$, up_0); Jump(up_0, up_1)



- State reuse will make the plan even more concise

  Moveback($down_1$, $down_0$)

Reached a solved state $down_0$.
Hence, the search stops!



Note that …

- State reuse and goal alternative can be applied independently of each other
  - In particular, goal alternative does NOT rely on state reuse to improve planning efficiency

- In our poster
  - we use the blocksworld example to show that goal alternative plays a more critical role than state reuse

Evaluation

- All problem instances belong to the benchmark domains of the IPC2008 FOND track
  - Blocksworld, Forest, Faults, and First-responders

- Goal
  - compare FIP, our planner that implements the Basic algorithm with the two extensions, against two state-of-the-art planners, MBP and Gamer
  - give each planner 1200 seconds to solve each problem instance

Evaluation 1: Problem Coverage

Domain	Gamer	MBP	FIP
blocksworld (30)	10	1	30
faults (55)	38	16	55
first-responders (100)	21	11	75
forest(90)	7	0	7
Total (275)	76	28	167

FIP solves more problems than Gamer and MBP within the time limit

Evaluation 2: Efficiency

Problem	Gamer		MBP	Basic		FIP	
	t	*s*	*t*	*t*	*s*	*t*	*s*
bw-1	38.748	10	*3556.517*	0.011	12	0.007	8
faults-8-8	1106.105	325	---	0.101	514	0.007	26
forest-2-8	0.638	56	---	0.007	56	0.008	56
forest-2-9	0.607	42	---	0.006	42	0.007	42
forest-2-10	0.927	44	---	0.007	44	0.008	44

- Comparing with the Basic Algorithm
  - FIP is on average more than 8 times faster than Basic
  - As the problem complexity increases, FIP could be more than 100 times faster than Basic
  - FIP's plans are 3.4 times smaller than Basic on average

- Comparing with MBP and Gamer
  - FIP is on average more than three orders of magnitude faster
  - FIP's plans are 2.8 times smaller than Gamer's

Evaluation 3: Which of the two extensions makes a more critical contribution?

FIP-SR-only: extends Basic with state reuse only

FIP-GA-only: extends Basic with goal alternative only

On average, FIP-GA-only runs more than 5 times faster than FIP-SR-only

FIP-GA-only creates plans that are 3.4 times smaller than FIP-SR-only.

Problem	FIP		FIP-SR-only		FIP-GA-only	
	t	*s*	*t*	*s*	*t*	*s*
bw-1	0.007	8	0.010	12	0.007	8
bw-2	0.006	7	0.008	10	0.005	7
bw-3	0.008	10	0.008	10	0.010	10
bw-4	0.011	14	0.012	16	0.013	14
bw-5	0.010	12	0.014	12	0.013	12
bw-6	0.009	10	0.010	10	0.011	10
bw-7	0.010	17	0.016	22	0.015	17
bw-8						
bw-9						
bw-10						
bw-12						
bw-25						
bw-30						
faults						
faults						
faults						
faults						
faults-10-7	0.007	32	0.960	2140	0.008	32
faults-10-10	0.009	32	1.101	2050	0.009	32
f-r-1-8	0.002	10	0.003	10	0.003	10
f-r-2-3	0.003	11	0.003	11	0.003	11
f-r-4-2	0.003	6	0.003	6	0.003	6
f-r-6-2	0.003	7	0.003	7	0.003	7
forest-2-5	0.008	56	0.007	56	0.009	56
forest-2-6	0.008	50	0.007	50	0.009	50
forest-2-7	0.008	44	0.007	44	0.009	44
forest-2-8	0.008	56	0.007	56	0.009	56
forest-2-9	0.007	42	0.006	42	0.008	42
forest-2-10	0.008	44	0.006	44	0.009	44

So, goal alternative plays a more crucial role than state reuse in improving planning efficiency and reducing plan size!

Summary

- Proposed two extensions to the Basic strong cyclic planning algorithm, goal alternative and state reuse

- FIP significantly outperforms state-of-the-art planners in terms of problem coverage, efficiency, and solution size.