Heuristic Search Algorithms and Markov Decision Processes

Rick Valenzano and Sheila McIlraith

UNIVERSITY OF TORONTO

Recap of Last Week

- Considered variants of sequential decision-making
  - Deterministic vs Non-Deterministic vs Stochastic
  - Fully Observable vs. Partially Observable
  - Model-based vs Model-free
  - Goal-seeking vs. Reward seeking

- Started with classical planning
  - Fully observable, deterministic, implicitly defined transition system, defined start state and goal tests

- Heuristic search-based planning
  - Looks at planning as graph search

Recap of Last Week

- Can use **Dijkstra's search**
  - Or incremental version, **Uniform-Cost Search**
- Uniform-cost search ignores the state information
  - Not practical
- Heuristic functions encode state information
  - Provides an estimate of the cost-to-go
  - Encodes domain information or automatically generated

This Week

- Hill climbing techniques
- The A* Algorithm
  - Completeness and optimality
- Greedy Best-First Search
- Weighted A*
  - Bounded suboptimality
- Markov Decision Processes
  - Stochastic state transitions
  - Rewards vs goals
  - Value Functions, Bellman equations

Employing Heuristics

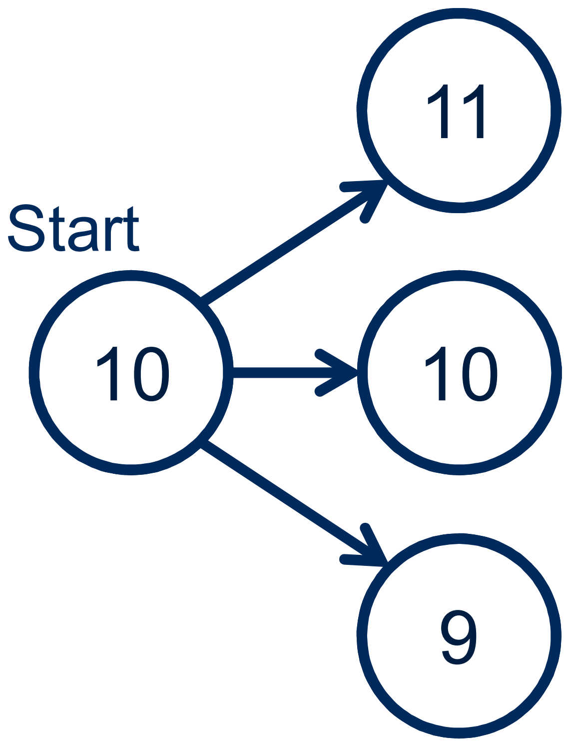
- Given a heuristic function h
  - What do we do with it?

Hill-Climbing

- Commit to the "best" child according to *h*

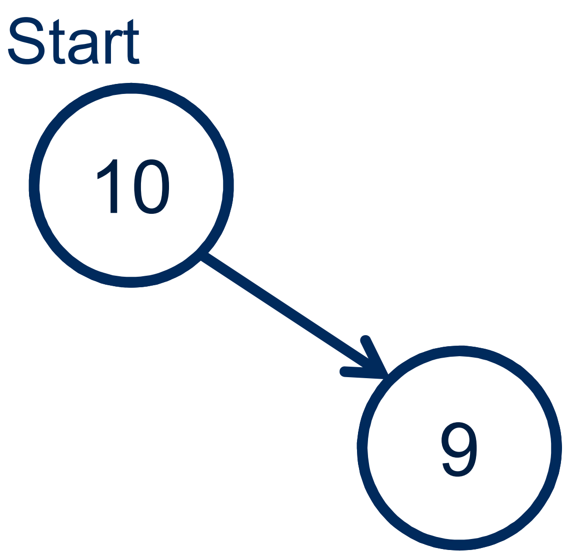
Hill-Climbing

- Commit to the "best" child according to *h*



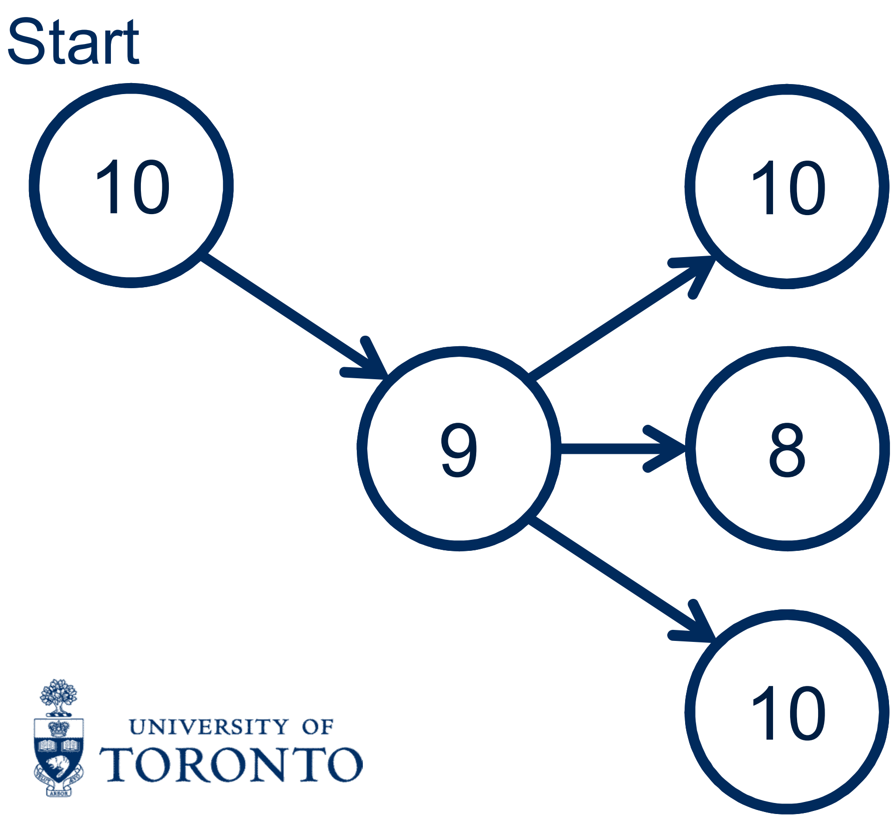
Hill-Climbing

- Commit to the "best" child according to *h*



Hill-Climbing

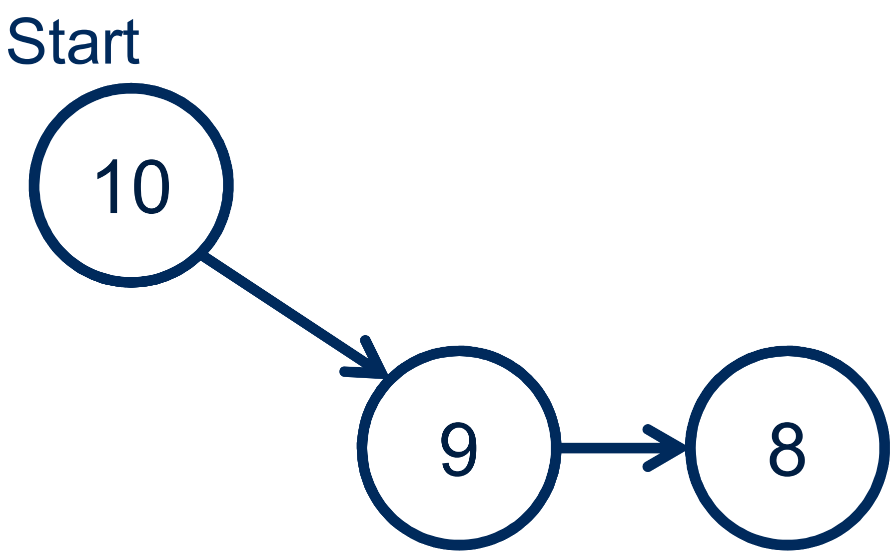
- Commit to the "best" child according to *h*



UNIVERSITY OF TORONTO

Hill-Climbing

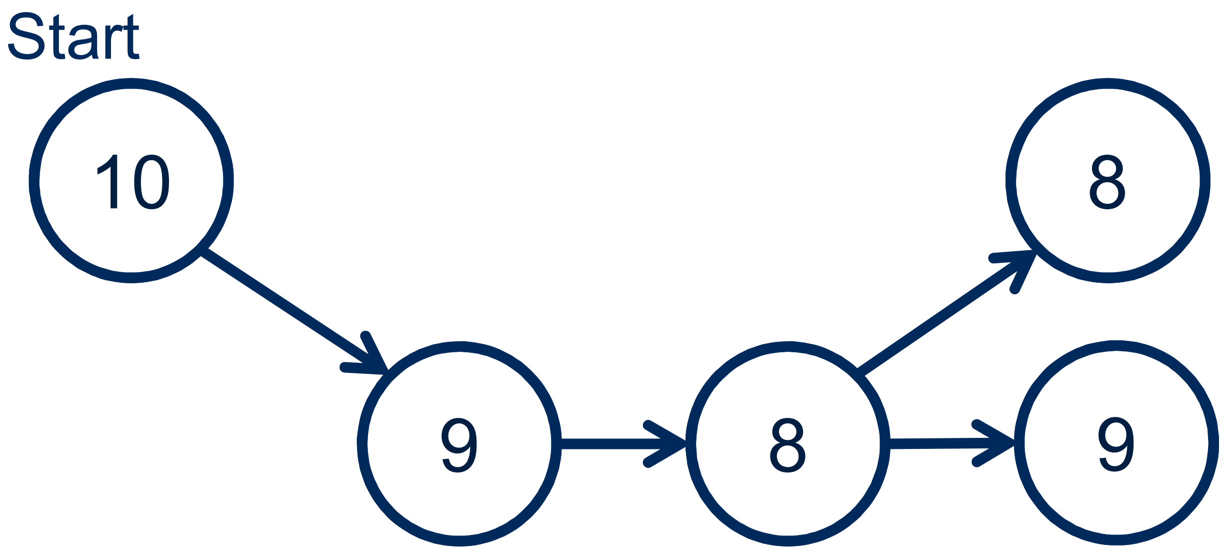
- Commit to the "best" child according to *h*



UNIVERSITY OF TORONTO

Hill-Climbing

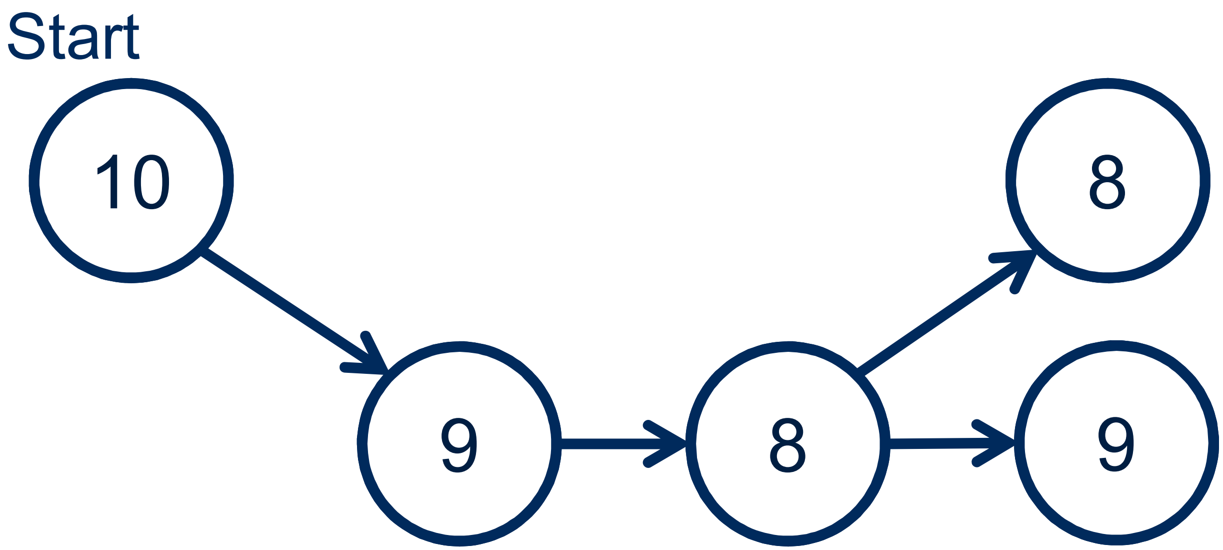
- Commit to the “best” child according to *h*



UNIVERSITY OF TORONTO

Hill-Climbing

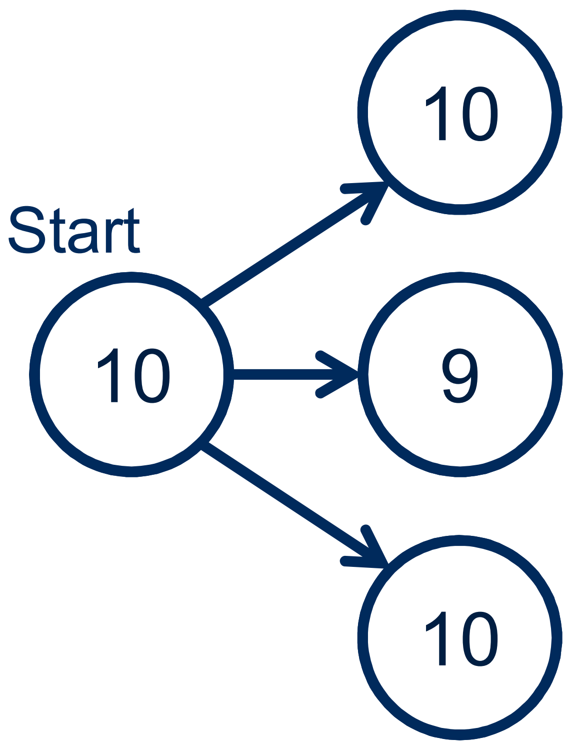
- What did we do now?



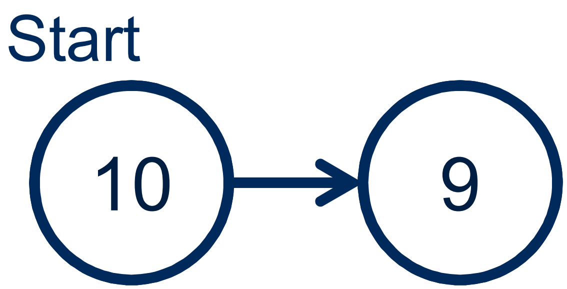
Hill-Climbing

- Multiple options
  – Pick "best of bad options"
  – Pick randomly
  – All kinds of local search strategies

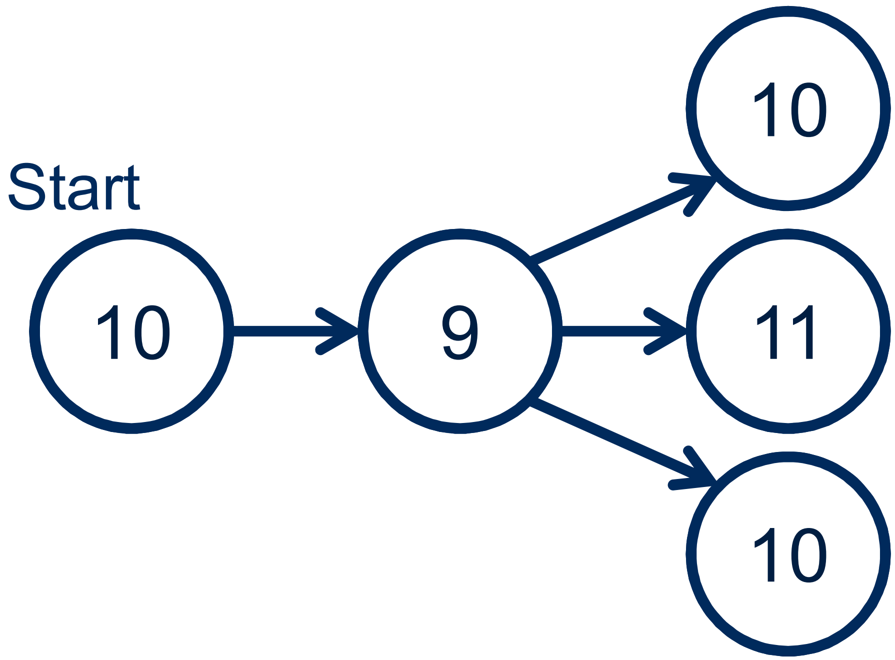
Enforced Hill-Climbing



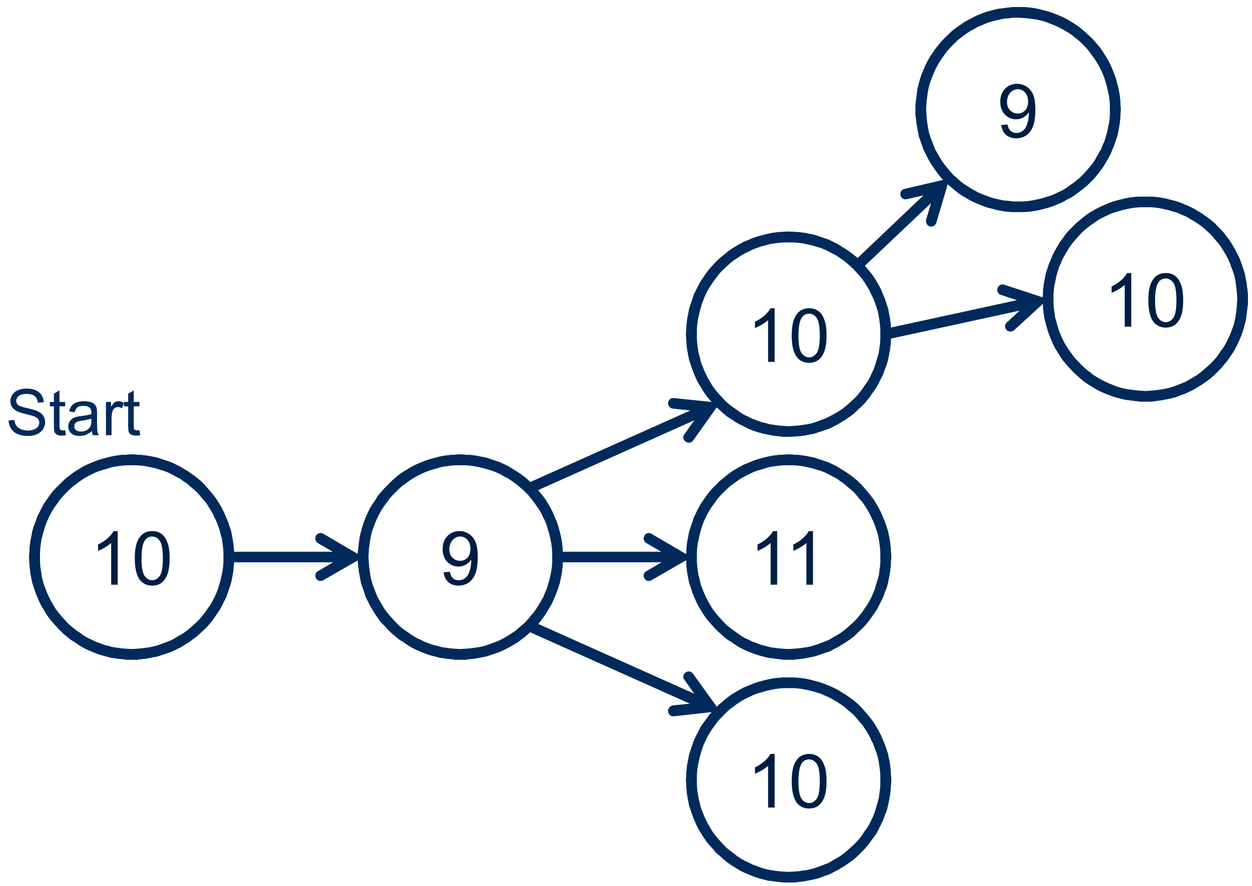
Enforced Hill-Climbing



Enforced Hill-Climbing

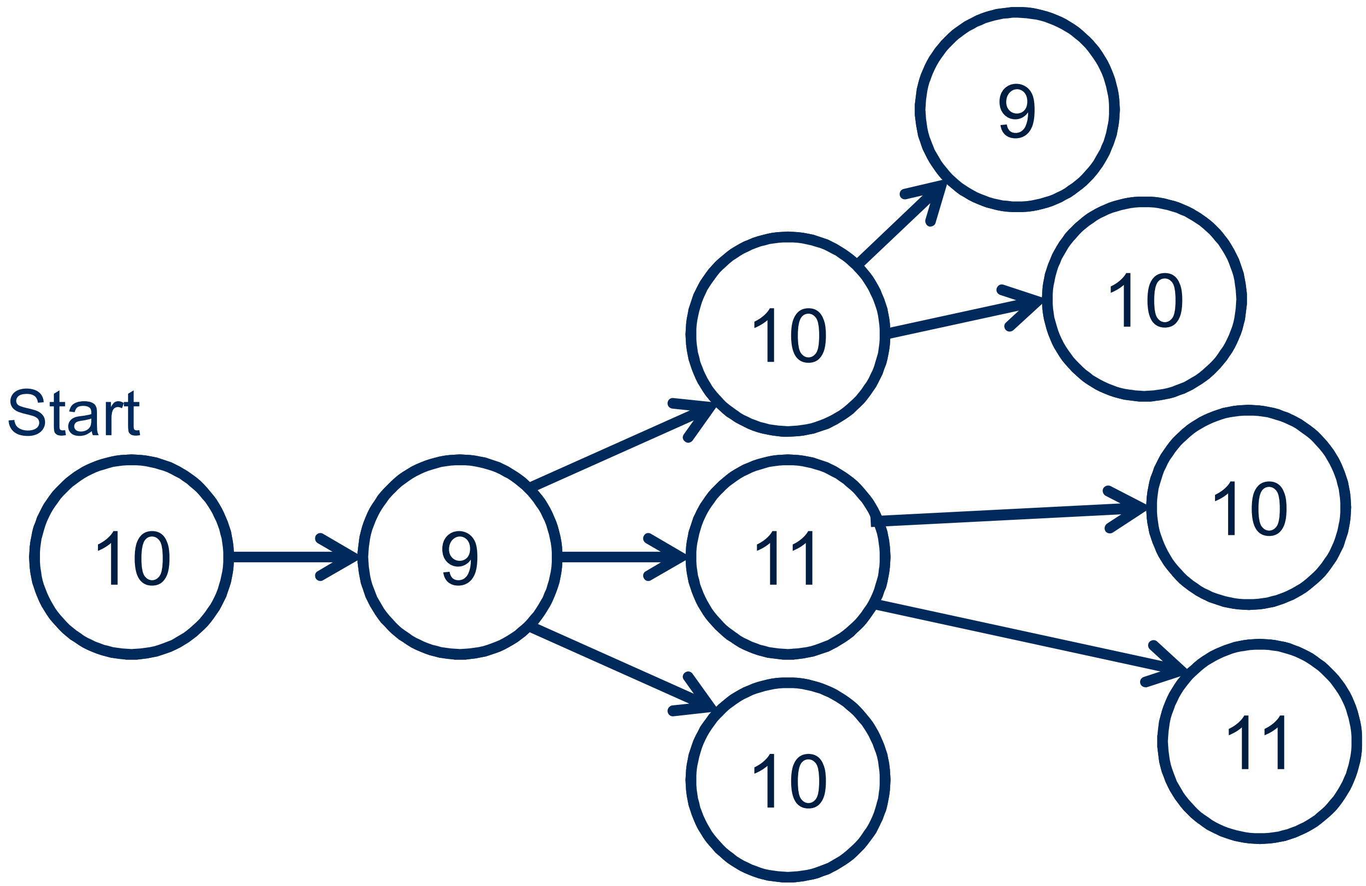


Enforced Hill-Climbing

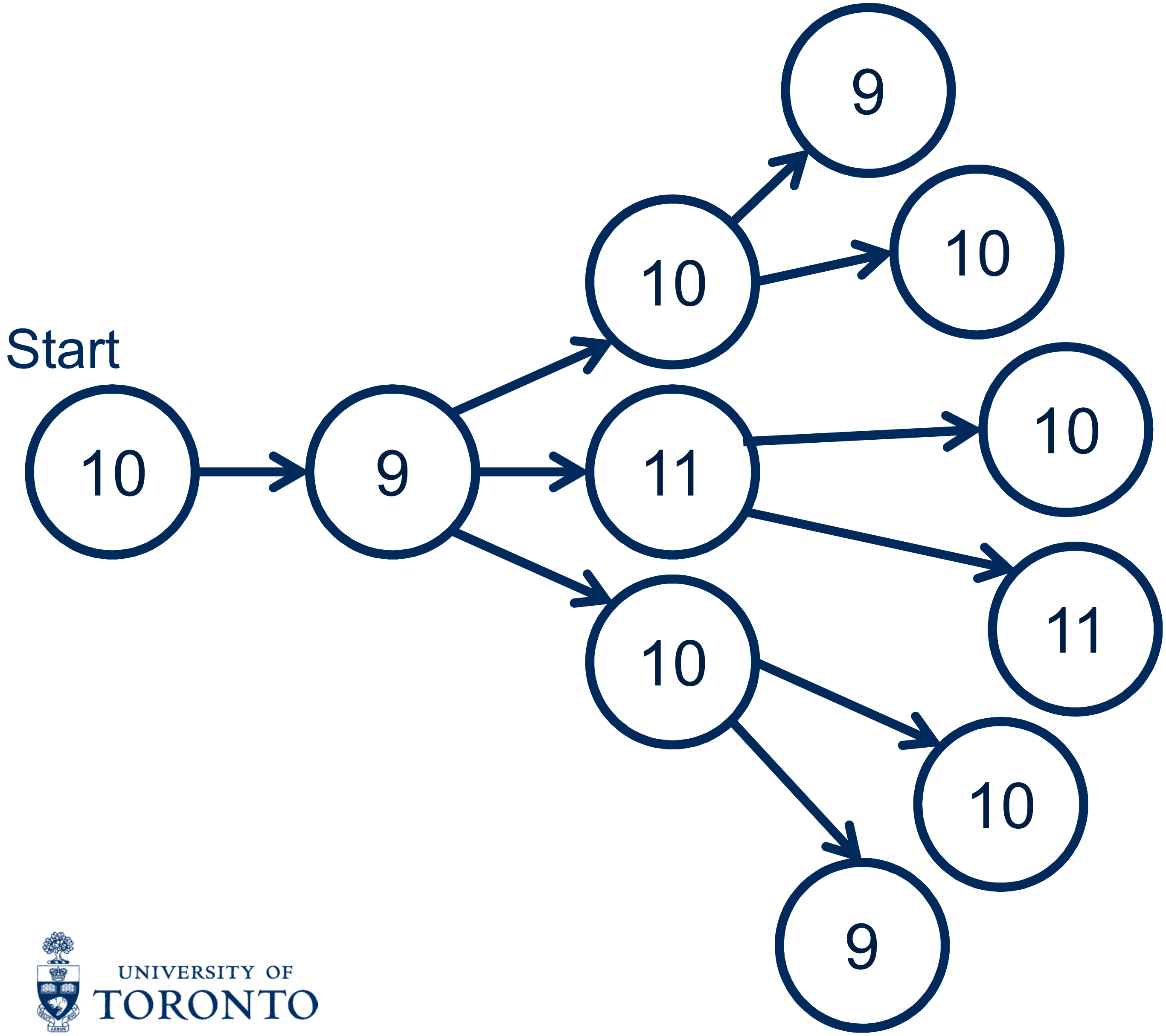


UNIVERSITY OF TORONTO

Enforced Hill-Climbing

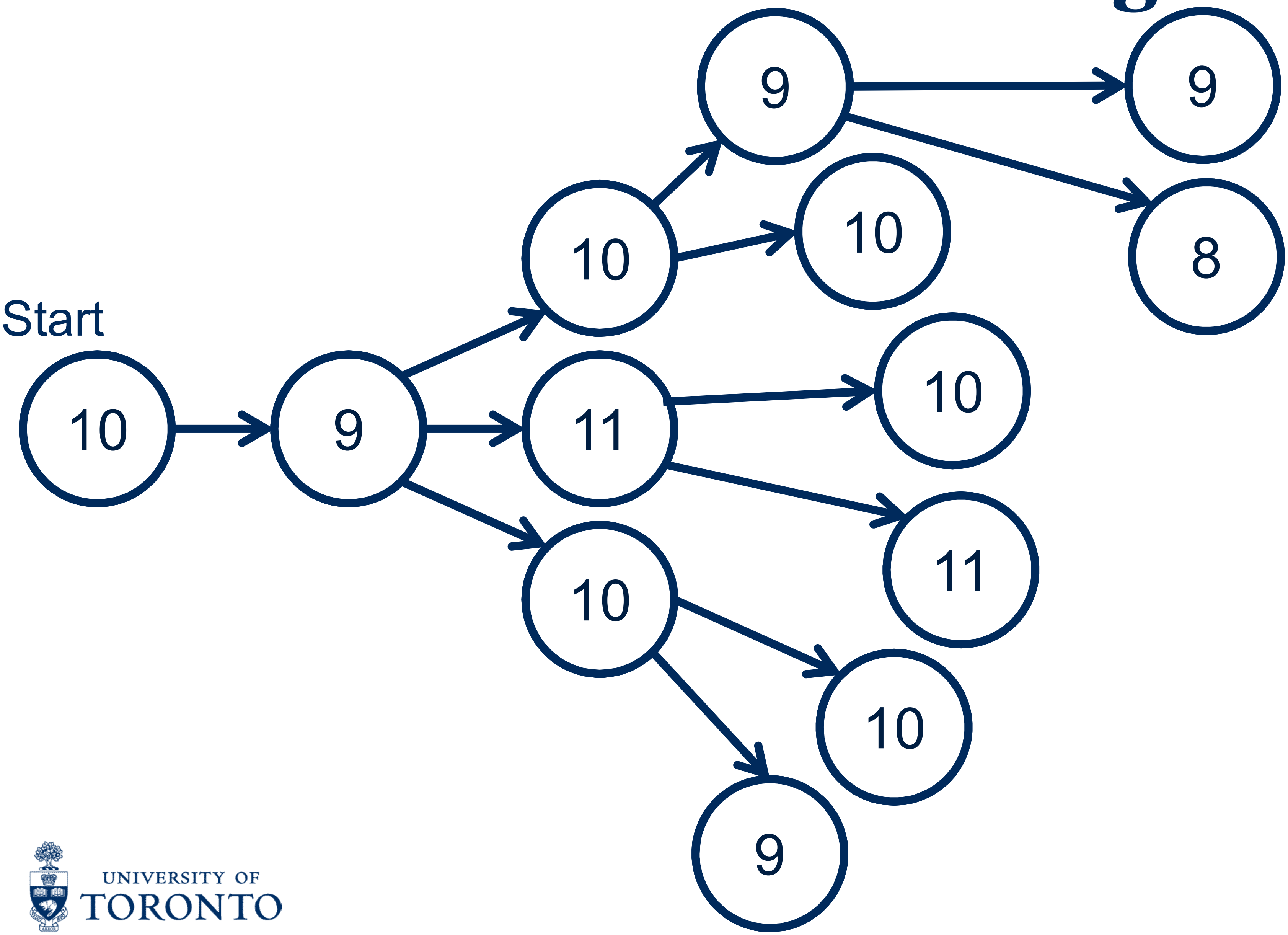


Enforced Hill-Climbing

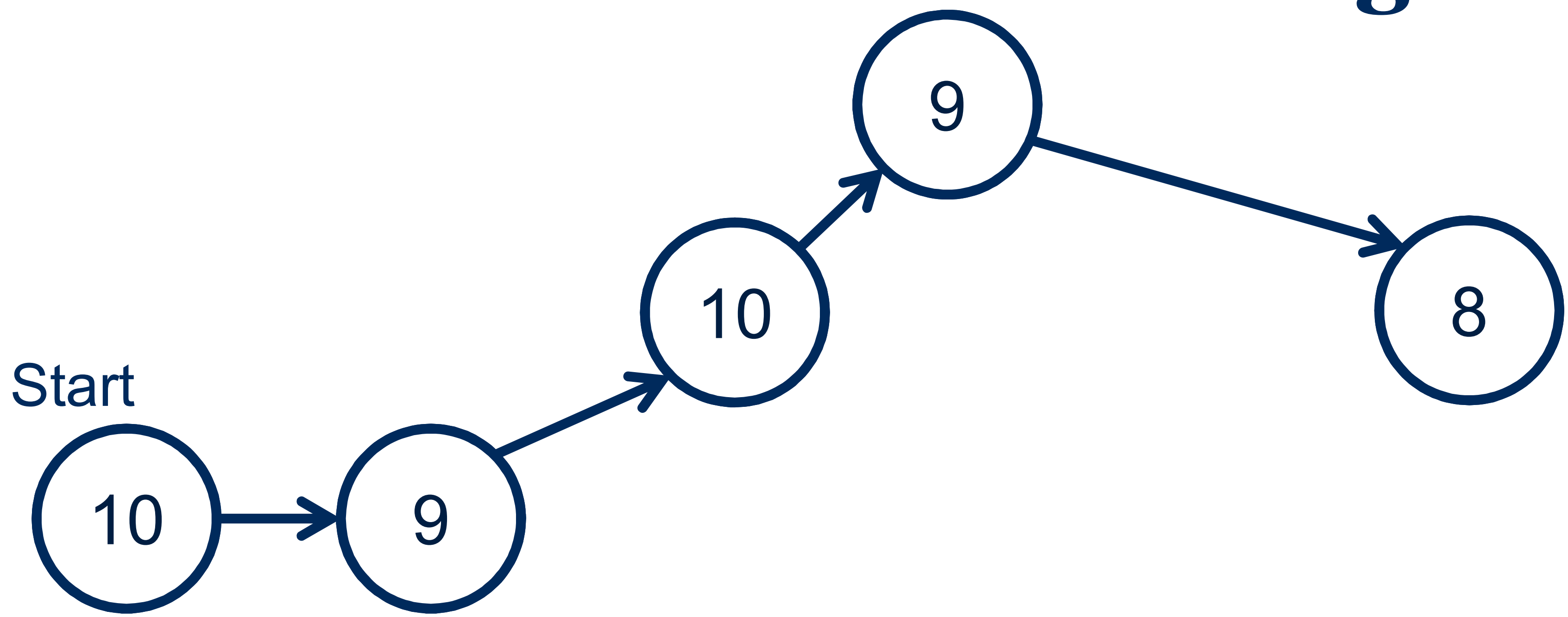


UNIVERSITY OF TORONTO

Enforced Hill-Climbing

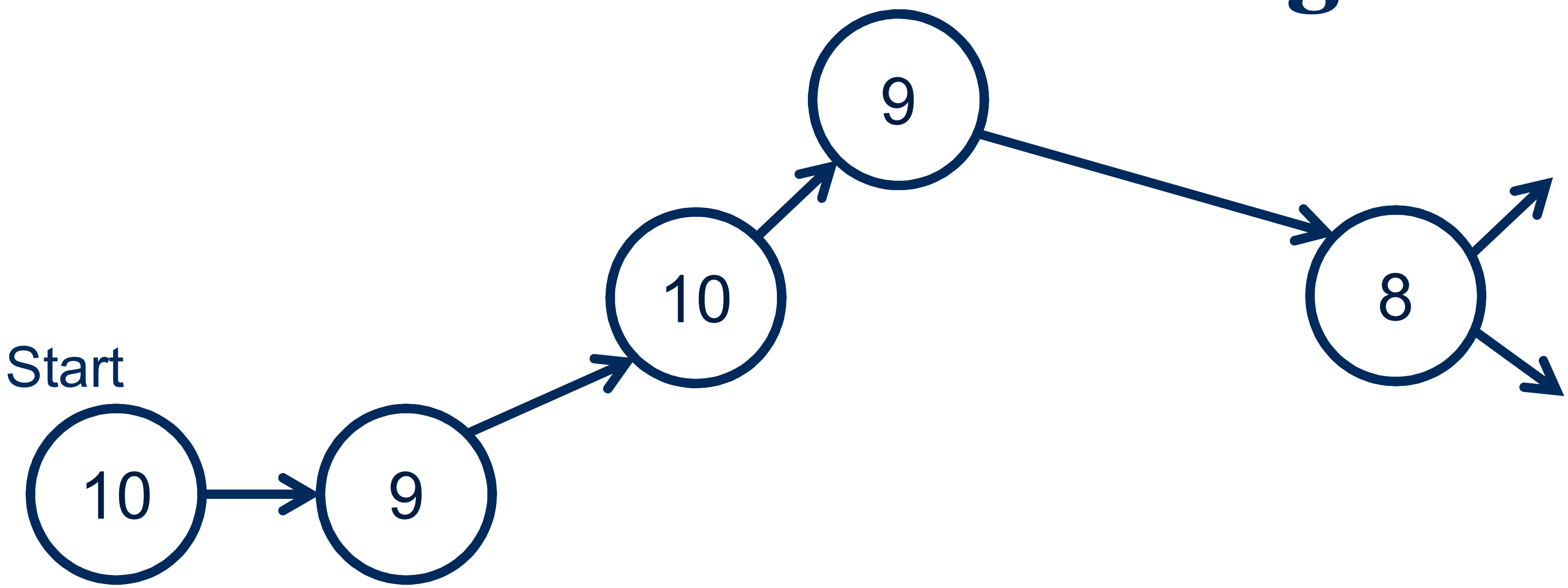


Enforced Hill-Climbing



UNIVERSITY OF TORONTO

Enforced Hill-Climbing



UNIVERSITY OF TORONTO

Search Algorithm Properties

Optimality

An solution found by the search algorithm is guaranteed to be optimal.

Completeness

The algorithm is guaranteed to find a solution to a given problem if one exists.

Search Algorithm Properties

Optimality

An solution found by the search algorithm is guaranteed to be optimal.

– Hill-climbing is not optimal.

Completeness

The algorithm is guaranteed to find a solution to a given problem if one exists.

– Hill-climbing is not complete.

Hill-Climbing

- So what is hill-climbing good for?

Uniform-Cost Search

Optimality

An solution found by the search algorithm is guaranteed to be optimal.

- Uniform-cost search is optimal

Completeness

The algorithm is guaranteed to find a solution to a given problem if one exists.

- Uniform-cost search is complete on finite state-spaces.

UNIVERSITY OF TORONTO

def UniformCostSearch(s_I):

$OPEN \leftarrow \{s_I\}, CLOSED \leftarrow \{\},$
$g(s_I) = 0, parent(s_I) = \emptyset$
while $OPEN \neq \{\}$:
  p ← $\text{argmin}_{\{s' \in OPEN\}} g(s')$
  if p is a goal, **return** path to p
  for $c \in children(p)$:
    if $c \notin OPEN \cup CLOSED$:
      $g(c) = g(p) + \kappa(p, c)$
      $parent(c) = p$
      $OPEN \leftarrow OPEN \cup \{c\}$
    else if $g(c) > g(p) + \kappa(p, c)$:
      $g(c) = g(p) + \kappa(p, c)$
      $parent(c) = p$
      if $c \in CLOSED$:
        $OPEN \leftarrow OPEN \cup \{c\}$
        $CLOSED \leftarrow CLOSED - \{c\}$
  $OPEN \leftarrow OPEN - \{p\}, CLOSED \leftarrow CLOSED \cup \{p\}$
return No solution exists

```
def UniformCostSearch(s_I):
    OPEN ← {s_I}, CLOSED ← {},
    g(s_I) = 0, parent(s_I) = ∅
    while OPEN ≠ {}:
        p ← argmin_{s′∈OPEN} g(s′)
        if p is a goal, return path to p
        for c ∈ children(p):
            if c ∉ OPEN ∪ CLOSED:
                g(c) = g(p) + κ(p, c)
                parent(c) = p
                OPEN ← OPEN ∪ {c}
            else if g(c) > g(p) + κ(p, c):
                g(c) = g(p) + κ(p, c)
                parent(c) = p
                if c ∈ CLOSED:
                    OPEN ← OPEN ∪ {c}
                    CLOSED ← CLOSED − {c}
        OPEN ← OPEN − {p}, CLOSED ← CLOSED ∪ {p}
    return No solution exists
```

```
def UniformCostSearch(s_I):
    OPEN ← {s_I}, CLOSED ← {},
    g(s_I) = 0, parent(s_I) = ∅
    while OPEN ≠ {}:
        p ← SelectNode(OPEN)
        if p is a goal, return path to p
        for c ∈ children(p):
            if c ∉ OPEN ∪ CLOSED:
                g(c) = g(p) + κ(p, c)
                parent(c) = p
                OPEN ← OPEN ∪ {c}
            else if g(c) > g(p) + κ(p, c):
                g(c) = g(p) + κ(p, c)
                parent(c) = p
                if c ∈ CLOSED:
                    OPEN ← OPEN ∪ {c}
                    CLOSED ← CLOSED − {c}
        OPEN ← OPEN − {p}, CLOSED ← CLOSED ∪ {p}
    return No solution exists
```

Open-Closed List Algorithms

- Open-Closed List (OCL) algorithms
  - Generalizes uniform-cost search
  - Allows for different ways of selecting nodes from OPEN

- Will use the heuristic function in SelectNode

Best-First Search

- Best-first search using an **evaluation function**

$$\Phi : \text{nodes} \rightarrow \mathbb{R}^{\geq 0}$$

- Defines the "value" of a node
  - Always selects the node with the lowest Φ-cost

def SelectNode($OPEN$):
    return $\text{argmin}_{\{n' \in OPEN\}} \Phi(n')$

UNIVERSITY OF TORONTO

Best-First Search

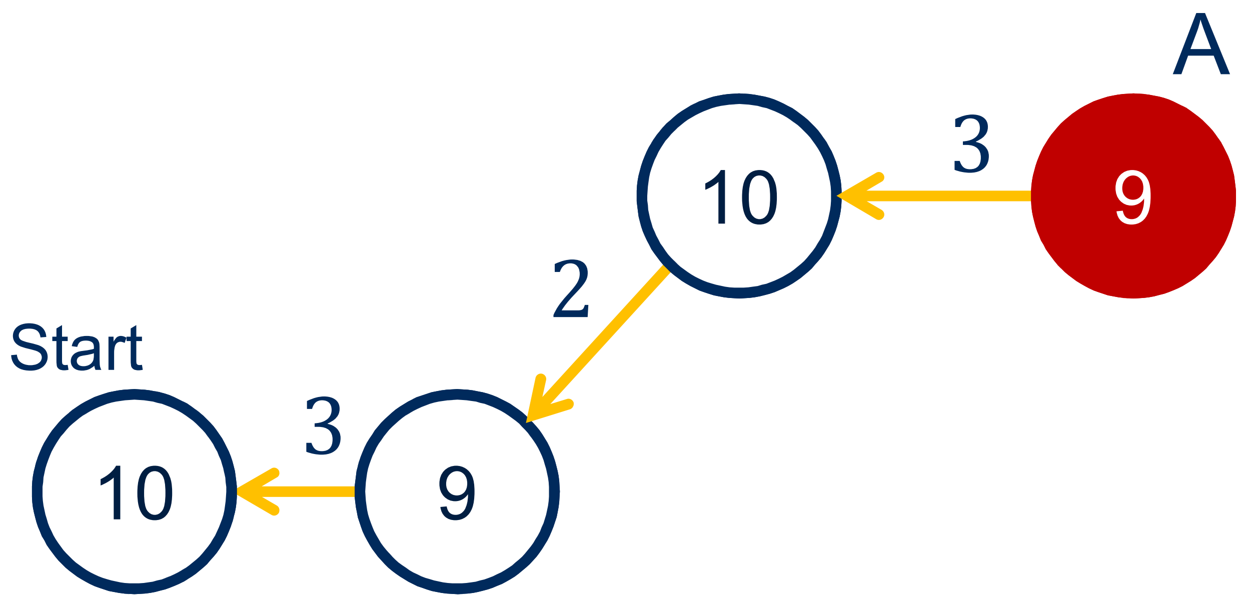
- Best-first search using an **evaluation function**

$$\Phi : \text{nodes} \rightarrow \mathbb{R}^{\geq 0}$$

def SelectNode($OPEN$):
    return $\text{argmin}_{\{n' \in OPEN\}} \Phi(n')$

- Uniform-cost search uses $\Phi(n) = g(n)$

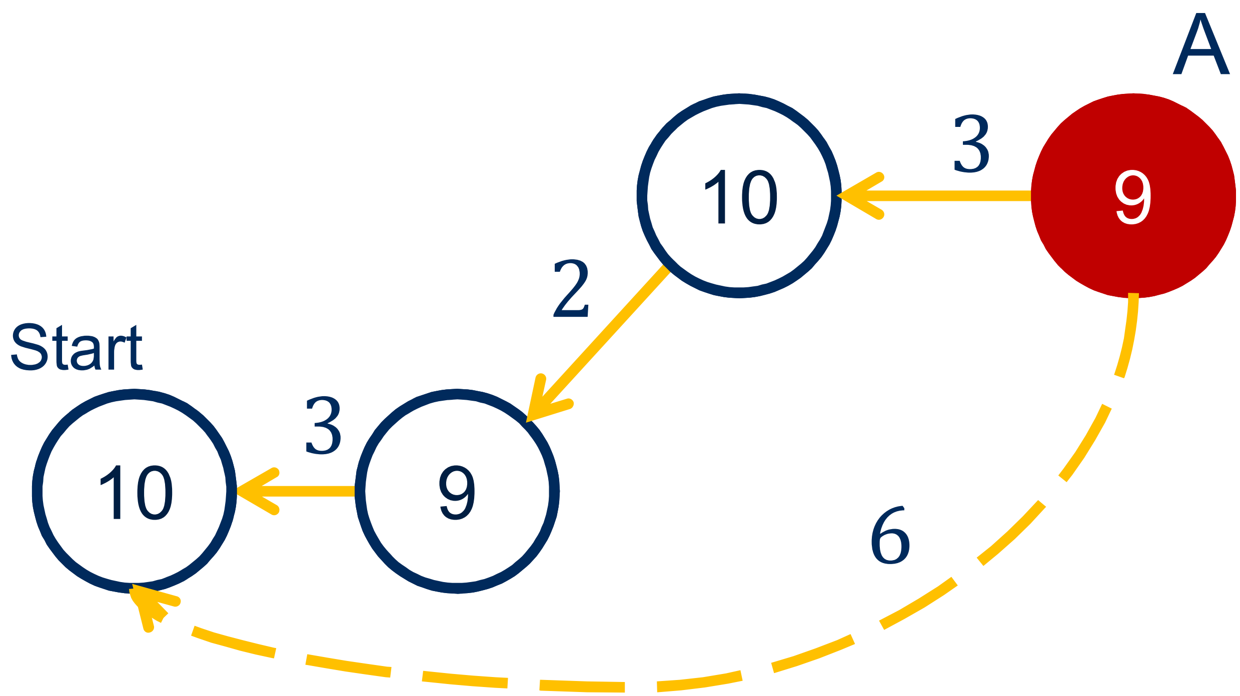
OCL Terminology



$g(\text{A}) = 8, h(\text{A}) = 9$

UNIVERSITY OF TORONTO

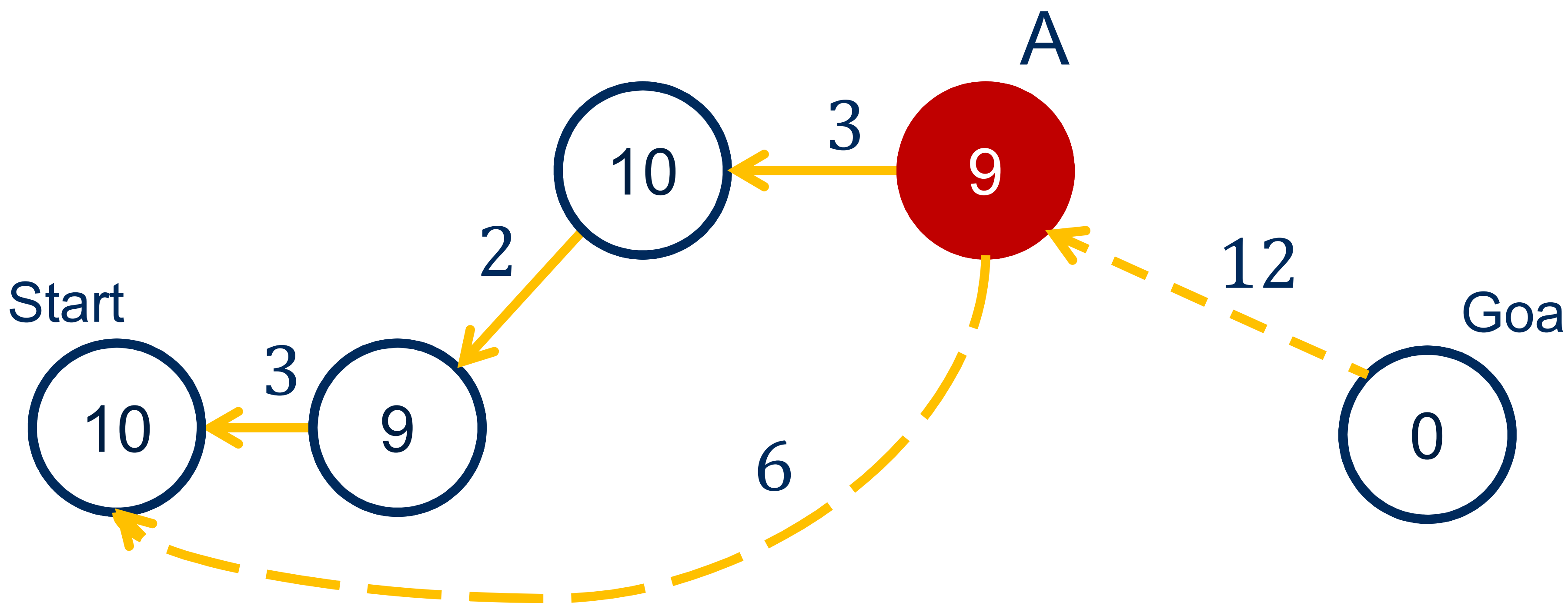
OCL Terminology



$g(\mathrm{A}) = 8, h(\mathrm{A}) = 9, g^*(A) = 6$

UNIVERSITY OF TORONTO

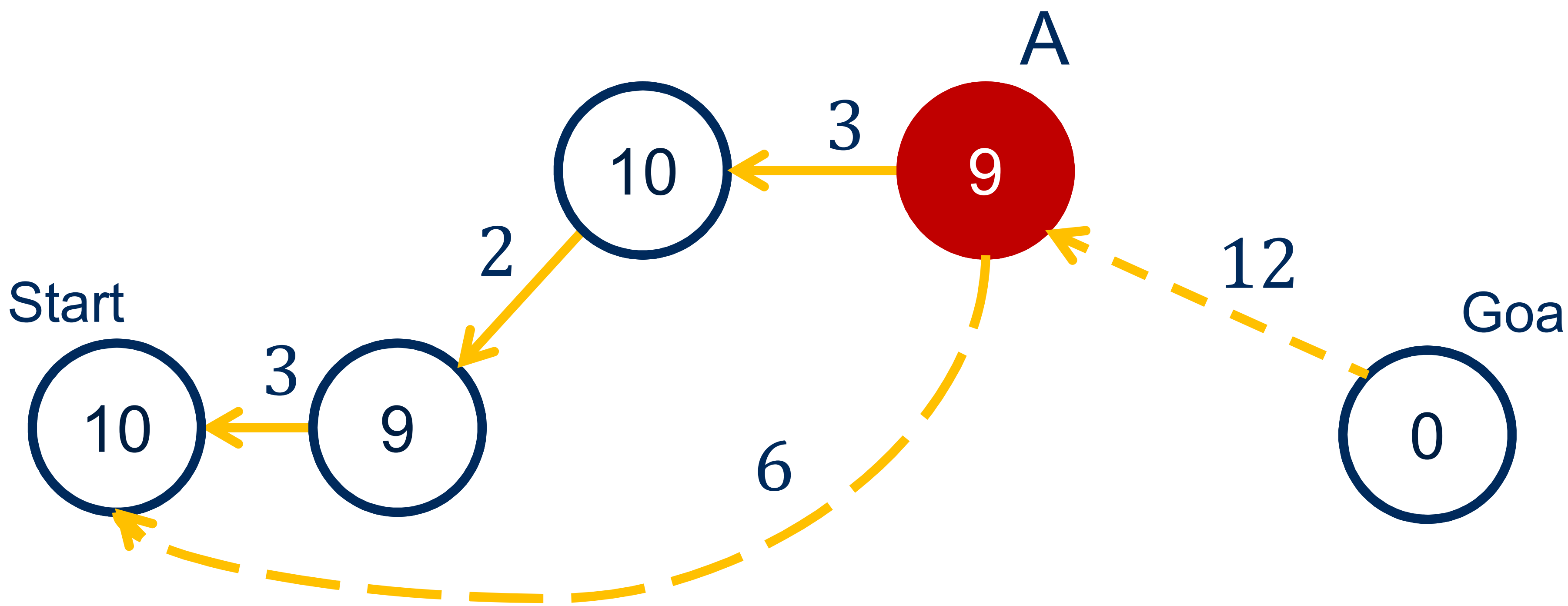
OCL Terminology



$g(\mathrm{A}) = 8, h(\mathrm{A}) = 9, g^*(A) = 6, h^*(\mathrm{A}) = 12$

UNIVERSITY OF TORONTO

OCL Terminology

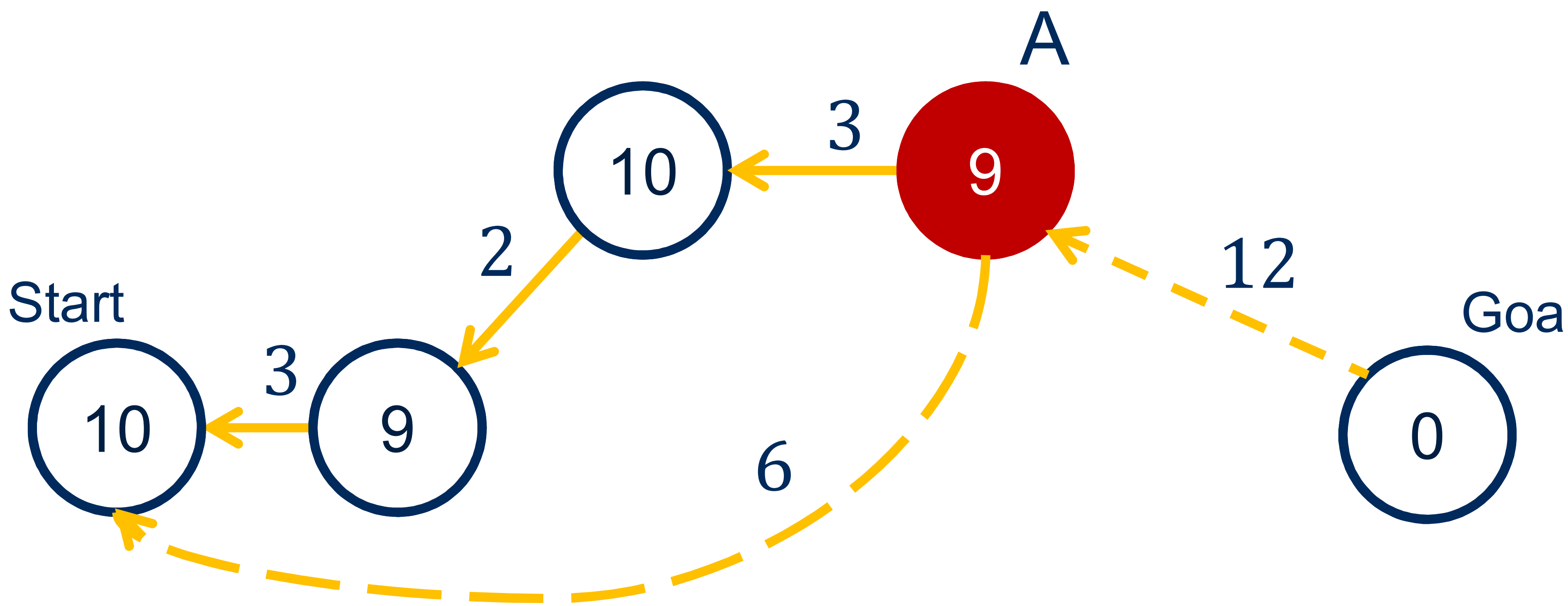


$g(\mathrm{A}) = 8, h(\mathrm{A}) = 9, g^*(A) = 6, h^*(\mathrm{A}) = 12$

C^* is the optimal solution path to the problem

UNIVERSITY OF TORONTO

OCL Terminology



$g(\mathrm{A}) = 8, h(\mathrm{A}) = 9, g^*(A) = 6, h^*(\mathrm{A}) = 12$

C^* is the optimal solution path to the problem

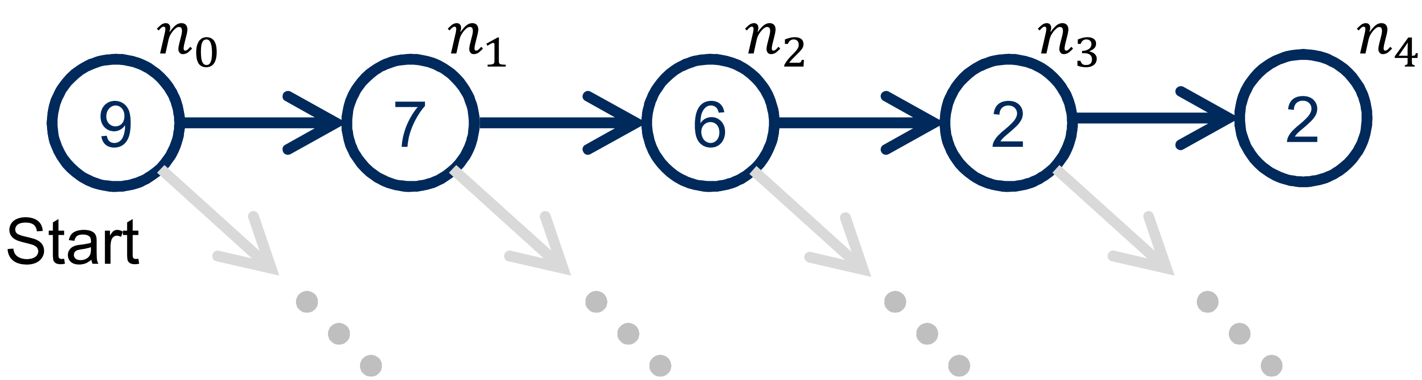
$C^* = 18$ if it passes through A

UNIVERSITY OF TORONTO

OCL Algorithms

Candidate Path Lemma

Let $P = [n_0, \ldots, n_k]$ be an optimal path to a given problem. Then at any time prior to the expansion of a goal node, there will be some node n_i from P in $OPEN$ with the optimal g-cost (ie. $g(n) = g^*(n)$).

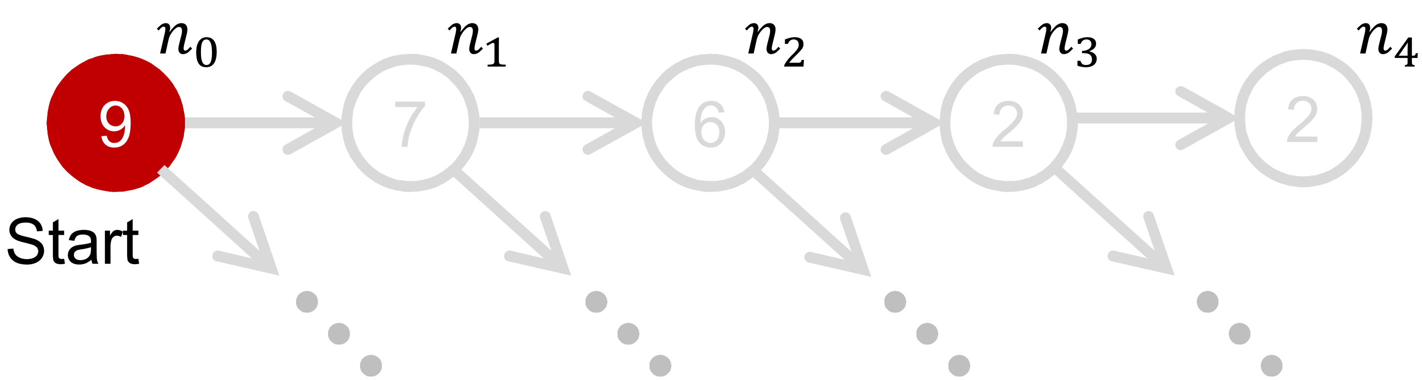


UNIVERSITY OF TORONTO

OCL Algorithms

Candidate Path Lemma

Let $P = [n_0, \ldots, n_k]$ be an optimal path to a given problem. Then at any time prior to the expansion of a goal node, there will be some node n_i from P in $OPEN$ with the optimal g-cost (ie. $g(n) = g^*(n)$).

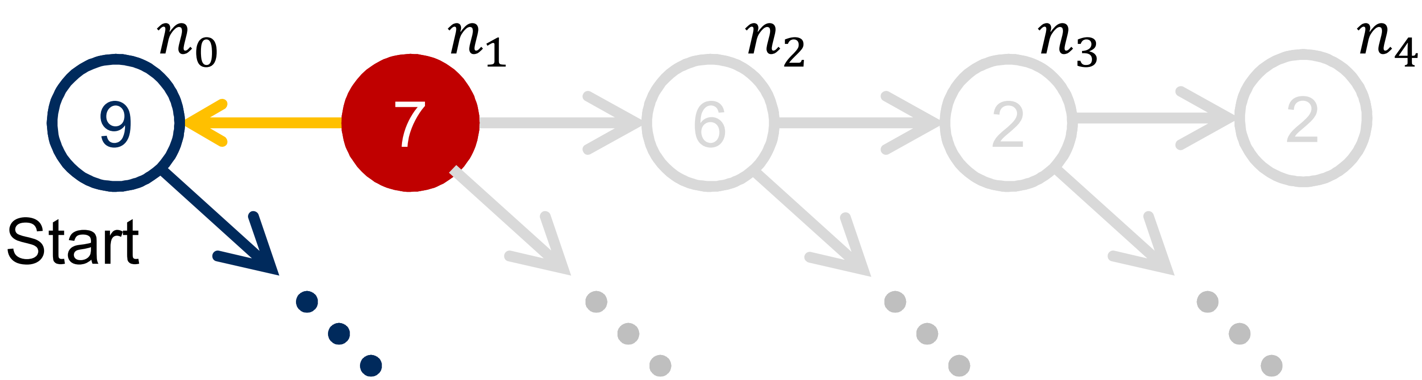


UNIVERSITY OF TORONTO

OCL Algorithms

Candidate Path Lemma

Let $P = [n_0, \dots, n_k]$ be an optimal path to a given problem. Then at any time prior to the expansion of a goal node, there will be some node n_i from P in $OPEN$ with the optimal g-cost (ie. $g(n) = g^*(n)$).

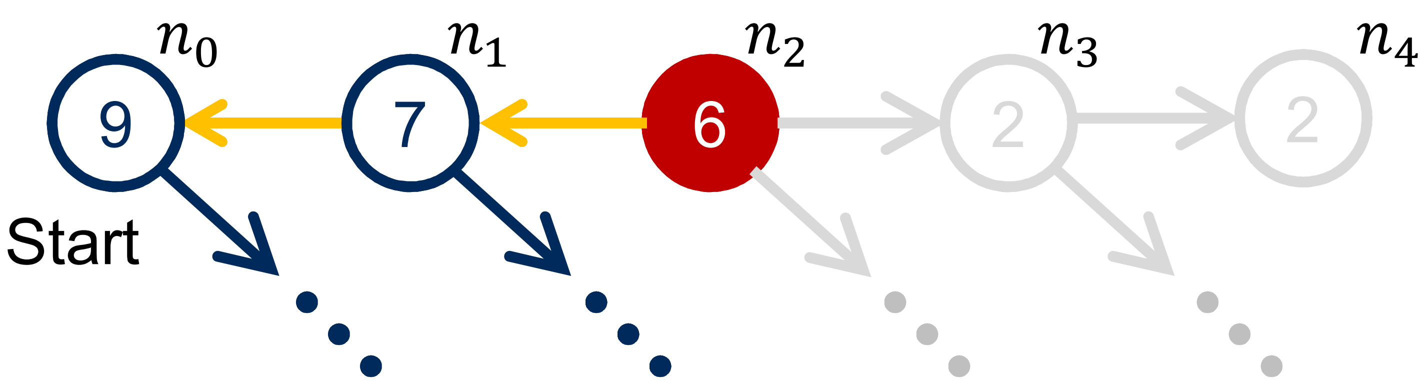


UNIVERSITY OF TORONTO

OCL Algorithms

Candidate Path Lemma

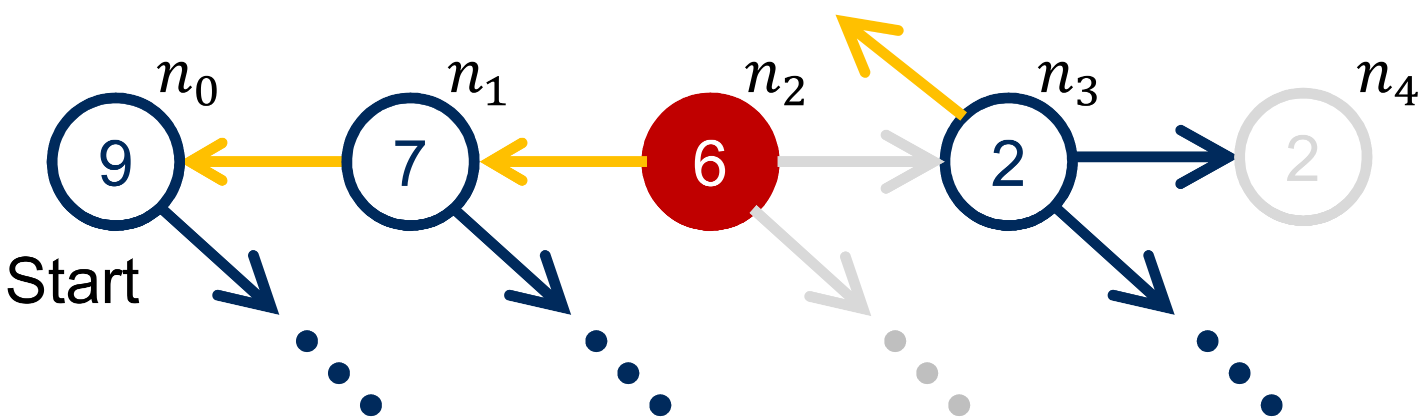
Let $P = [n_0, \ldots, n_k]$ be an optimal path to a given problem. Then at any time prior to the expansion of a goal node, there will be some node n_i from P in $OPEN$ with the optimal g-cost (ie. $g(n) = g^*(n)$).



OCL Algorithms

Candidate Path Lemma

Let $P = [n_0, \ldots, n_k]$ be an optimal path to a given problem. Then at any time prior to the expansion of a goal node, there will be some node n_i from P in $OPEN$ with the optimal g-cost (ie. $g(n) = g^*(n)$).



def OCL(s_I):

$OPEN \leftarrow \{s_I\}, CLOSED \leftarrow \{\},$
$g(s_I) = 0, parent(s_I) = \emptyset$
while $OPEN \neq \{\}$:
  p ← SelectNode($OPEN$)
  if p is a goal, **return** path to p
  for $c \in children(p)$:
    if $c \notin OPEN \cup CLOSED$:
      $g(c) = g(p) + \kappa(p, c)$
      $parent(c) = p$
      $OPEN \leftarrow OPEN \cup \{c\}$
    else if $g(c) > g(p) + \kappa(p, c)$:
      $g(c) = g(p) + \kappa(p, c)$
      $parent(c) = p$
      if $c \in CLOSED$:
        $OPEN \leftarrow OPEN \cup \{c\}$
        $CLOSED \leftarrow CLOSED - \{c\}$
  $OPEN \leftarrow OPEN - \{p\}, CLOSED \leftarrow CLOSED \cup \{p\}$
return No solution exists

def OCL(s_I):

$OPEN \leftarrow \{s_I\}, CLOSED \leftarrow \{\},$

$g(s_I) = 0, parent(s_I) = \emptyset$

while $OPEN \neq \{\}$:

p $\leftarrow$ SelectNode($OPEN$)

if p is a goal, **return** path to p

for $c \in children(p)$:

if $c \notin OPEN \cup CLOSED$:

$g(c) = g(p) + \kappa(p, c)$

$parent(c) = p$

$OPEN \leftarrow OPEN \cup \{c\}$

else if $g(c) > g(p) + \kappa(p, c)$:

$g(c) = g(p) + \kappa(p, c)$

$parent(c) = p$

if $c \in CLOSED$:

Node Reopening → $OPEN \leftarrow OPEN \cup \{c\}$

$CLOSED \leftarrow CLOSED - \{c\}$

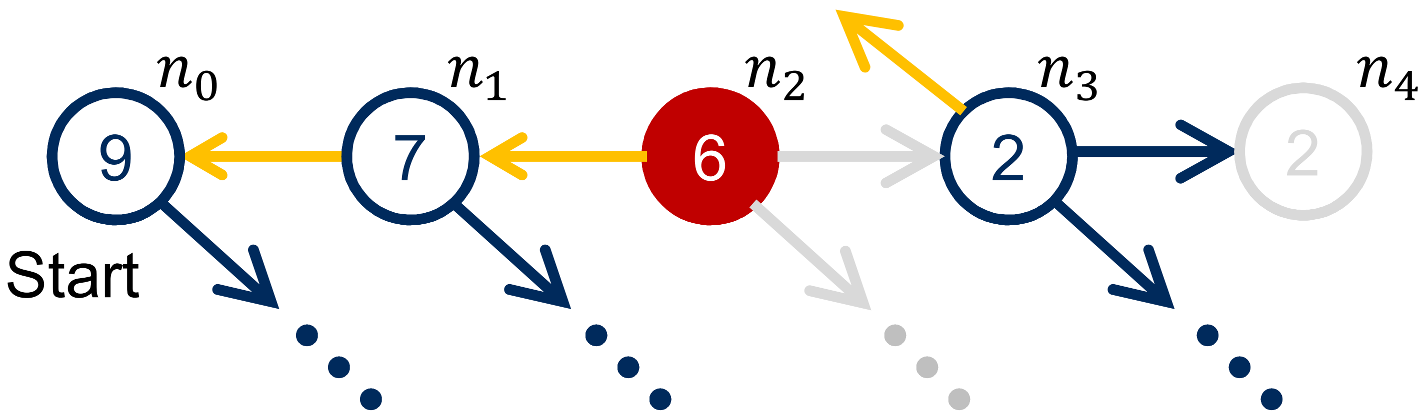
$OPEN \leftarrow OPEN - \{p\}, CLOSED \leftarrow CLOSED \cup \{p\}$

return No solution exists

OCL Algorithms

Candidate Path Lemma

Let $P = [n_0, \dots, n_k]$ be an optimal path to a given problem. Then at any time prior to the expansion of a goal node, there will be some node n_i from P in $OPEN$ with the optimal g-cost (ie. $g(n) = g^*(n)$).

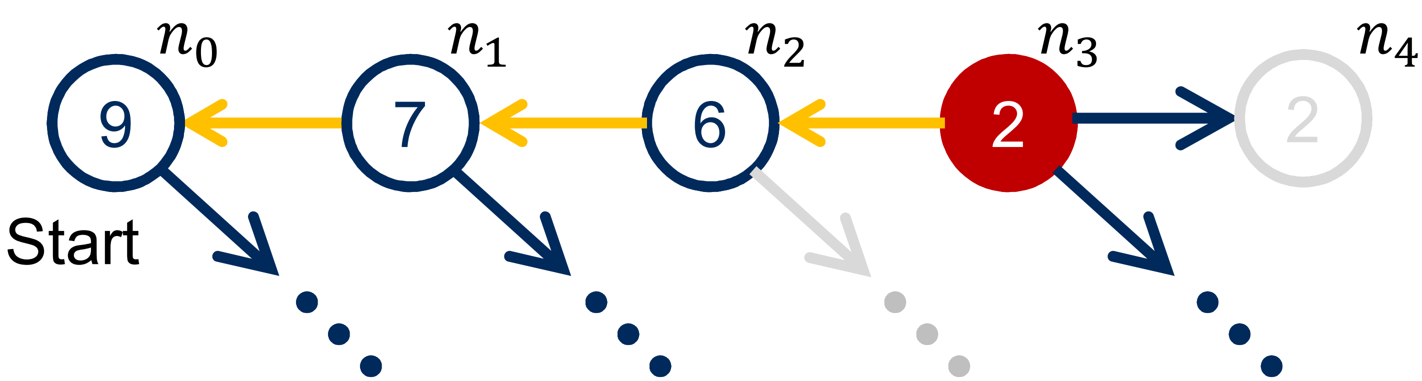


UNIVERSITY OF TORONTO

OCL Algorithms

Candidate Path Lemma

Let $P = [n_0, \ldots, n_k]$ be an optimal path to a given problem. Then at any time prior to the expansion of a goal node, there will be some node n_i from P in $OPEN$ with the optimal g-cost (ie. $g(n) = g^*(n)$).



UNIVERSITY OF TORONTO

def OCL(s_I):

$OPEN \leftarrow \{s_I\}, CLOSED \leftarrow \{\}$,

$g(s_I) = 0, parent(s_I) = \emptyset$

while $OPEN \neq \{\}$:

p ← SelectNode($OPEN$)

if p is a goal, **return** path to p

for $c \in children(p)$:

if $c \notin OPEN \cup CLOSED$:

$g(c) = g(p) + \kappa(p, c)$

$parent(c) = p$

$OPEN \leftarrow OPEN \cup \{c\}$

else if $g(c) > g(p) + \kappa(p, c)$:

$g(c) = g(p) + \kappa(p, c)$

$parent(c) = p$

if $c \in CLOSED$:

$OPEN \leftarrow OPEN \cup \{c\}$

$CLOSED \leftarrow CLOSED - \{c\}$

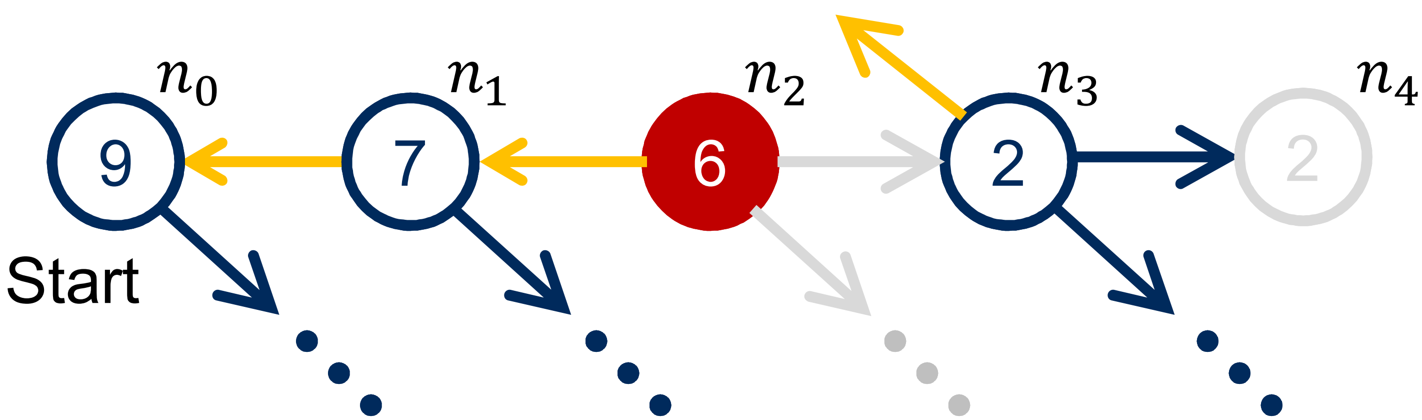
$OPEN \leftarrow OPEN - \{p\}, CLOSED \leftarrow CLOSED \cup \{p\}$

return No solution exists

OCL Algorithms

Candidate Path Lemma

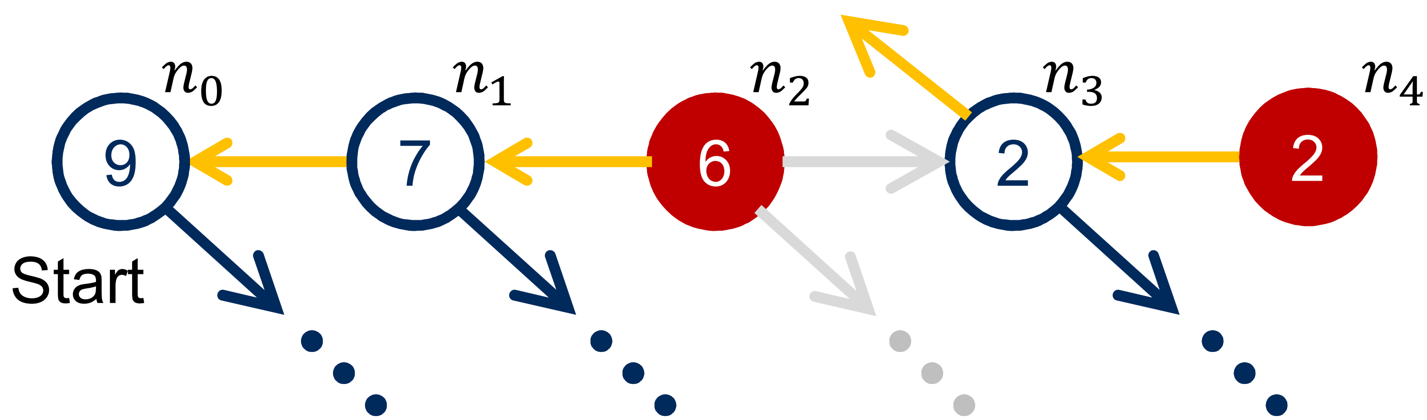
Let $P = [n_0, \ldots, n_k]$ be an optimal path to a given problem. Then at any time prior to the expansion of a goal node, there will be some node n_i from P in $OPEN$ with the optimal g-cost (ie. $g(n) = g^*(n)$).



OCL Algorithms

Candidate Path Lemma

Let $P = [n_0, \ldots, n_k]$ be an optimal path to a given problem. Then at any time prior to the expansion of a goal node, there will be some node n_i from P in $OPEN$ with the optimal g-cost (ie. $g(n) = g^*(n)$).



OCL Algorithms

OCL Completeness

Any OCL algorithm is complete on any solvable problem with a finite state-space.

Proof Sketch

1. The candidate path lemma ensures that *OPEN* can never become empty before a goal state is expanded.

UNIVERSITY OF TORONTO

OCL Algorithms

OCL Completeness

Any OCL algorithm is complete on any solvable problem with a finite state-space.

Proof Sketch

1. The candidate path lemma ensures that *OPEN* can never become empty before a goal state is expanded.

2. There are a finite number of paths to any node, so every node can only be re-expanded a finite number of times.

The A* Algorithm

- Best-first search using an **evaluation function**

$$\Phi : \text{nodes} \rightarrow \mathbb{R}^{\geq 0}$$

def SelectNode($OPEN$):
    return $\text{argmin}_{\{n' \in OPEN\}} \Phi(n')$

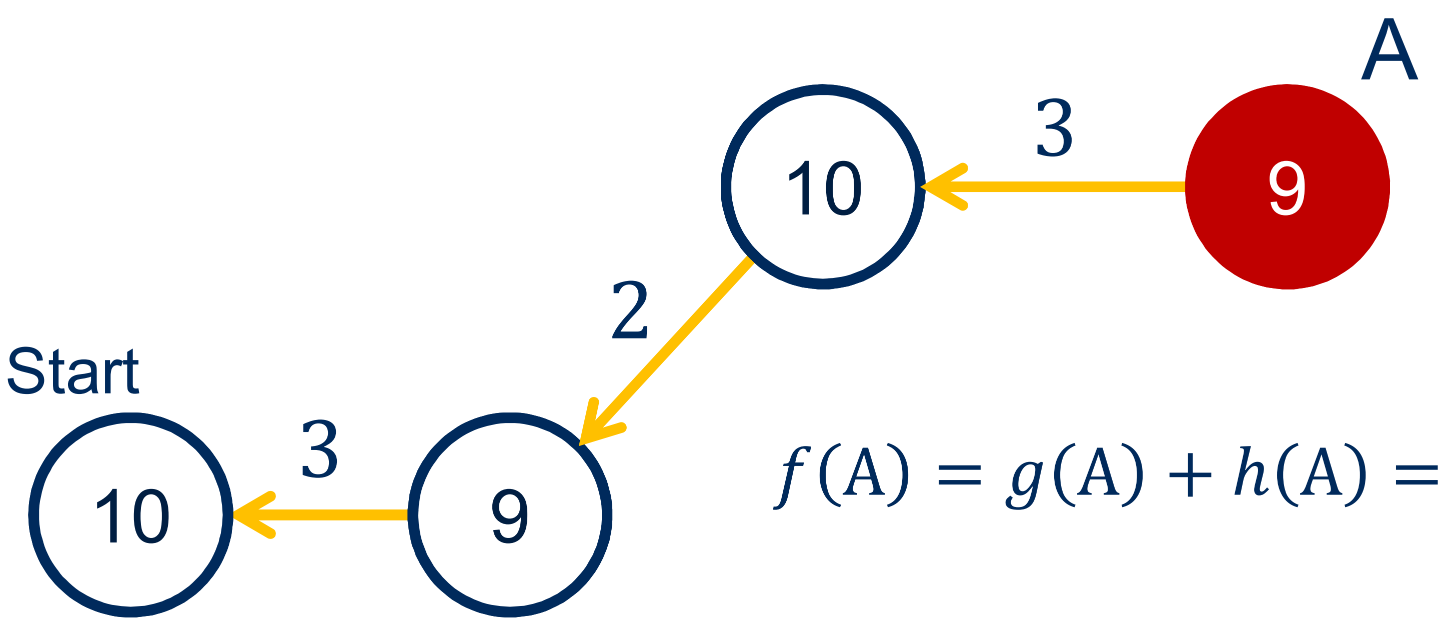
- A* uses $\Phi(n) = f(n) = g(n) + h(n)$

UNIVERSITY OF TORONTO

The A* Algorithm

def SelectNode($OPEN$):

  return $\text{argmin}_{\{n' \in OPEN\}}\ \text{g}(n') + h(n')$



$f(\text{A}) = g(\text{A}) + h(\text{A}) = 8 + 9 = 17$

$f(\text{A})$ is an estimate of the cost of the solution path through A

UNIVERSITY OF TORONTO

Heuristic Admissibility

- A*'s optimality relies on **admissibility**
  - Ensures the heuristic never overestimates the cost to go
  - "One-sided error"

Heuristic Admissibility

Heuristic h is **admissible** if $h(n) \leq h^*(n)$ for all n.

Optimality of A*

Optimality of A*

If the heuristic being used is admissible, then any solution found by A* will be optimal.

Optimality of A*

Optimality of A*

If the heuristic being used is admissible, then any solution found by A* will be optimal.

Proof Sketch.

By contradiction, show that a goal state along a suboptimal solution cannot be expanded before all the nodes along the optimal solution path.

Optimality of Uniform-Cost Search

Optimality of Uniform-Cost Search

Uniform-cost search will only find optimal solutions.

Optimality of Uniform-Cost Search

Optimality of Uniform-Cost Search

Uniform-cost search will only find optimal solutions.

Proof Sketch

Uniform-cost search is equivalent to A* using the heuristic h such that $h(n) = 0$ for all n.

UNIVERSITY OF TORONTO

Using the Heuristic to Prune

Avoiding Node Expansions

If the heuristic being used is admissible, then A* will not expand any nodes for which $f(n) > C^*$.

Using the Heuristic to Prune

Avoiding Node Expansions

If the heuristic being used is admissible, then A* will not expand any nodes for which $f(n) > C^*$.

Proof Sketch.

Before a goal is found, there will always be a node n' from the optimal solution path on *OPEN* such that

$$\begin{aligned} f(n') = g(n') + h(n') &= g^*(n') + h(n') \\ &\leq g^*(n') + h^*(n') = C^* \end{aligned}$$

UNIVERSITY OF TORONTO

A* vs. Uniform-Cost Search

- Uniform-cost search will not expand any n such that

$$g(n) > C^*$$

- A* may be able to expand fewer unique states than uniform-cost search due to heuristic pruning

- But what about re-expansions?

UNIVERSITY OF TORONTO

```
def OCL(s_I):
    OPEN ← {s_I}, CLOSED ← {},
    g(s_I) = 0, parent(s_I) = ∅
    while OPEN ≠ {}:
        p ← SelectNode(OPEN)
        if p is a goal, return path to p
        for c ∈ children(p):
            if c ∉ OPEN ∪ CLOSED:
                g(c) = g(p) + κ(p, c)
                parent(c) = p
                OPEN ← OPEN ∪ {c}
            else if g(c) > g(p) + κ(p, c):
                g(c) = g(p) + κ(p, c)
                parent(c) = p
                if c ∈ CLOSED:
                    OPEN ← OPEN ∪ {c}
                    CLOSED ← CLOSED − {c}
        OPEN ← OPEN − {p}, CLOSED ← CLOSED ∪ {p}
    return No solution exists
```

Node Reopening

Heuristic Consistency

Heuristic Consistency

Heuristic h is **consistent** if for any pair of node p and c, where c is a child of p, the following holds:

$$h(p) \leq h(c) + \kappa(p, c)$$

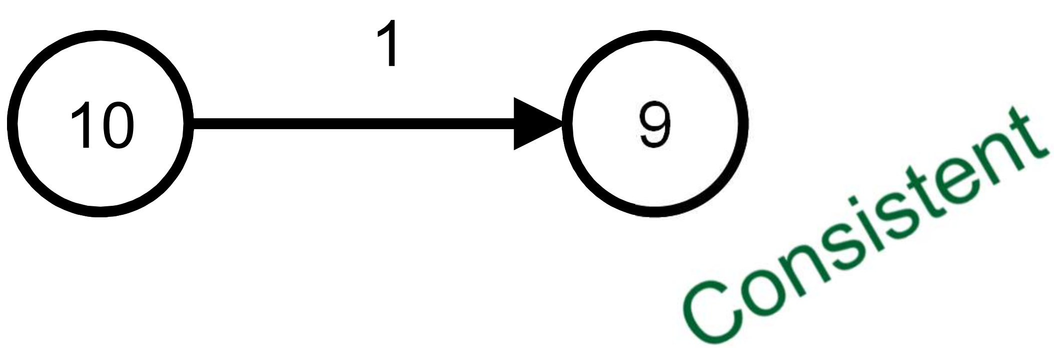
UNIVERSITY OF TORONTO

Heuristic Consistency

Heuristic Consistency

Heuristic h is **consistent** if for any pair of node p and c, where c is a child of p, the following holds:

$$h(p) \leq h(c) + \kappa(p, c)$$

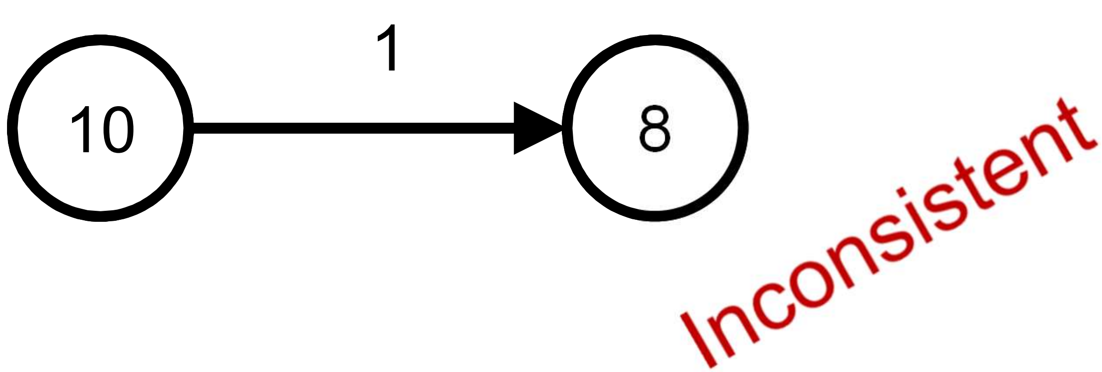


Heuristic Consistency

Heuristic Consistency

Heuristic h is **consistent** if for any pair of node p and c, where c is a child of p, the following holds:

$$h(p) \leq h(c) + \kappa(p, c)$$

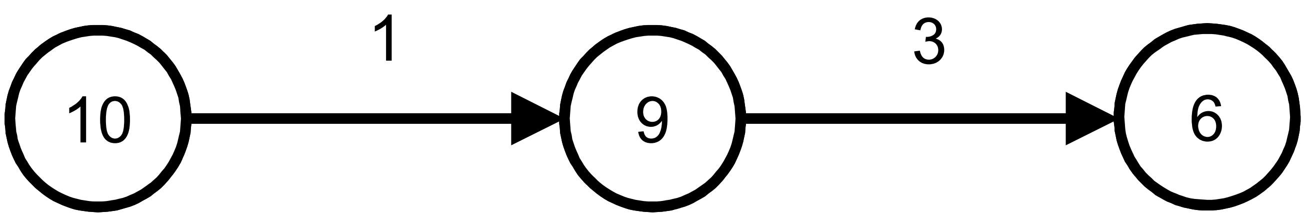


UNIVERSITY OF TORONTO

Heuristic Consistency

- Consistency guarantees a heuristic version of the triangle inequality:

$$h(p) \leq h(d) + \kappa(p, c) + \kappa(c, d)$$



UNIVERSITY OF TORONTO

Heuristic Consistency

Re-expansion Theorem

If the heuristic being used by A* is consistent, then A* will never **reopen** a node.

Heuristic Consistency

Re-expansion Theorem

If the heuristic being used by A* is consistent, then A* will never **reopen** a node.

or alternatively

If the heuristic being used by A* is consistent, then whenever A* expands a node n, $g(n) = g^*(n)$

A* vs. Uniform-Cost Search

- A* will do at least as much pruning as UCS
- If the heuristic is consistent, no node will be expanded more than once
- If the heuristic allows some pruning, A* should be faster than UCS

The A* Algorithm

- Recall proof that A* is optimal
- Similar argument shows A* expands every node with $f(n) < C^*$ where C^* is the optimal solution cost
  - This is how it proves that the optimal solution has been found
- Proving optimality of a found solution path can make A* prohibitively expensive

UNIVERSITY OF TORONTO

Weighted A* (WA*)

- Weighted A* is also a best-first search algorithm

def SelectNode($OPEN$):
return $\text{argmin}_{\{n' \in OPEN\}} \Phi(n')$

- WA* uses $\Phi(n) = f_w(n) = g(n) + w \cdot h(n)$
  - The **weight** w is an parameter where $w \geq 1$

UNIVERSITY OF TORONTO

Weighted A* (WA*)

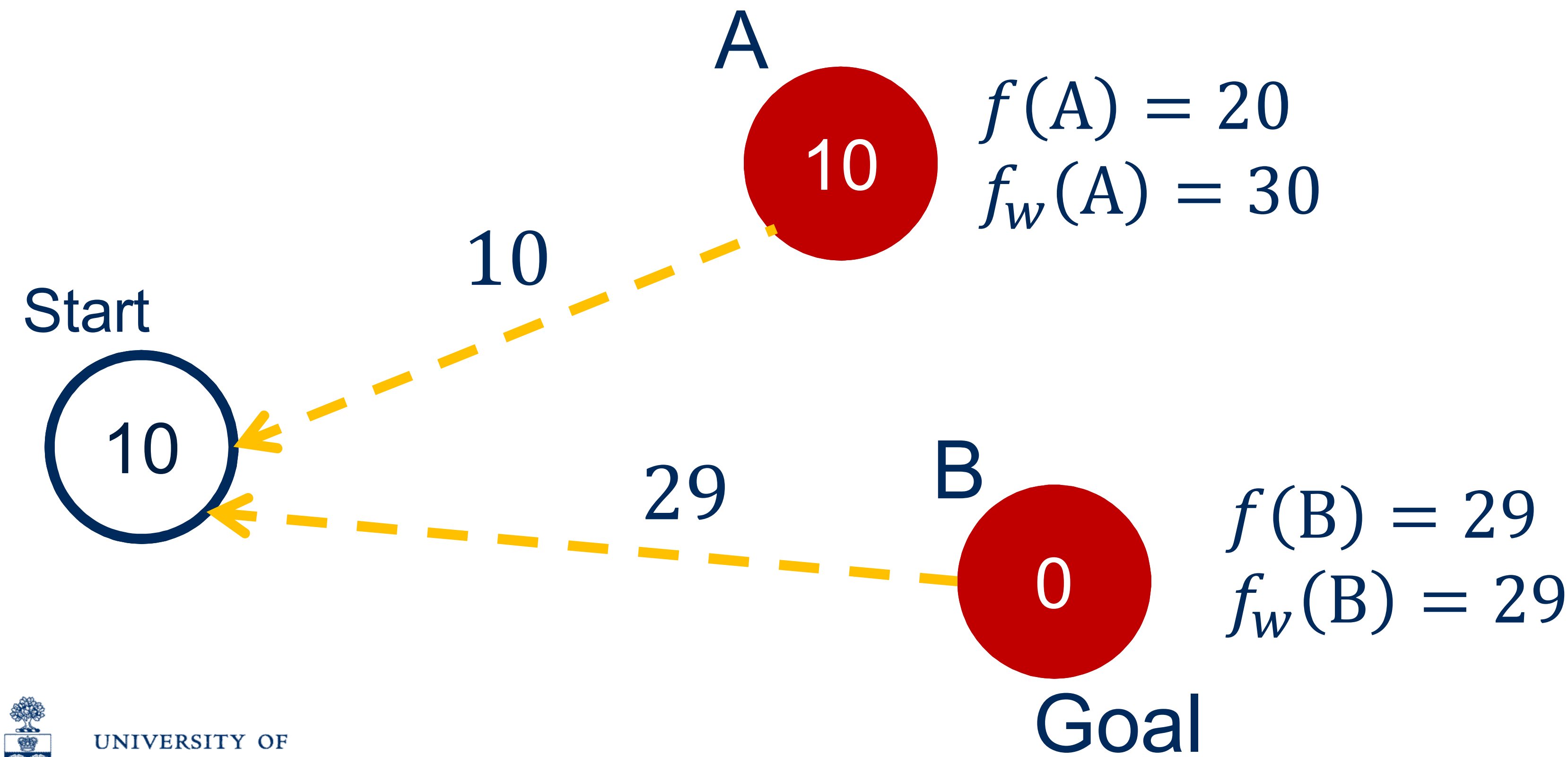
$$f_w(n) = g(n) + w \cdot h(n)$$

- The weight impacts the relative importance of the h-cost and the g-cost
  - h-cost dominates the evaluation for large w
  - WA* becomes greedier on h as w increases

Weighted A* (WA*)

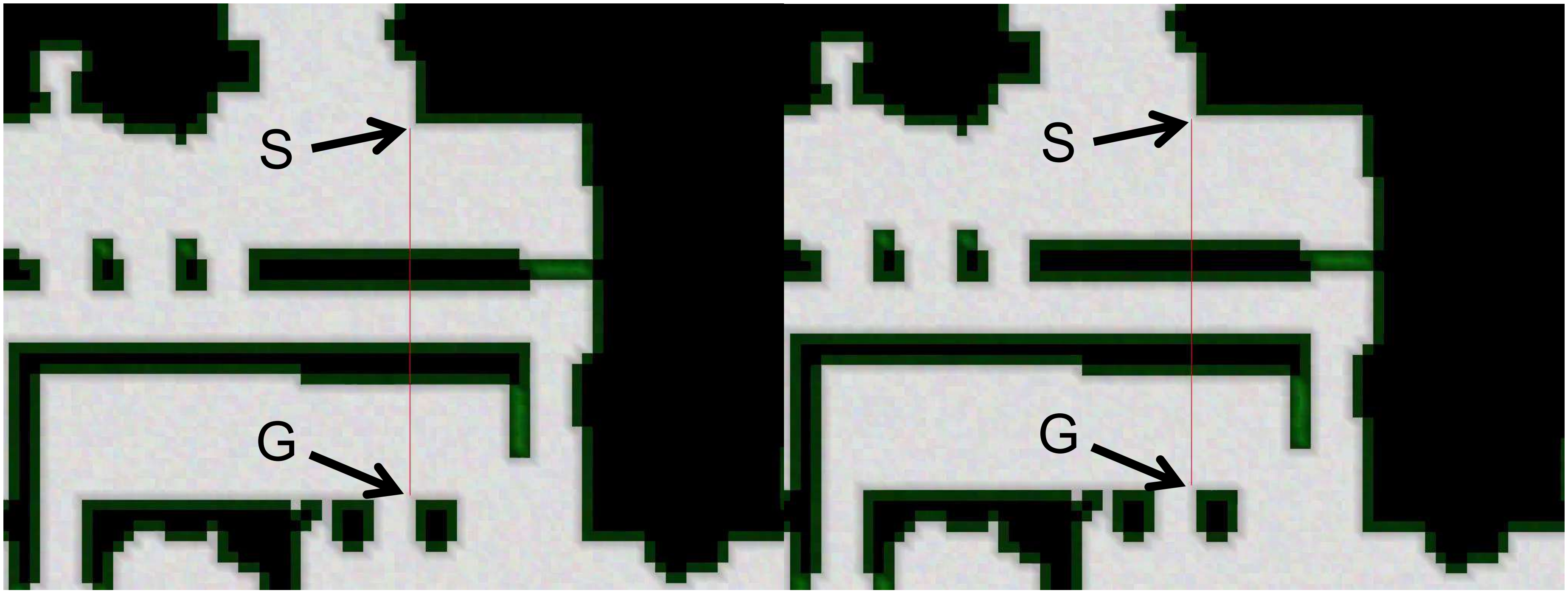
$$f_w(n) = g(n) + w \cdot h(n)$$

$w = 2$



UNIVERSITY OF TORONTO

A* vs. WA*



UNIVERSITY OF TORONTO

Weighted A* Properties

Optimality

Weighted A* is not an optimal algorithm.

Completeness

Weighted A* is a complete algorithm.

UNIVERSITY OF TORONTO

Weighted A* Suboptimality

Bounded Suboptimality

If the heuristic being used is admissible, then any solution found by WA* will cost no more than $w \cdot C^*$.

Weighted A* Suboptimality

Bounded Suboptimality

If the heuristic being used is admissible, then any solution found by WA* will cost no more than $w \cdot C^*$.

Proof Sketch.

This is ensured by the f_w and the way nodes are selected for expansion.

Greedy Best-First Search

- **Greedy Best-First Search (GBFS)** is WA* "in the limit"
  - Still a best-first search, but maximally greedy on h

def SelectNode($OPEN$):
    return $\text{argmin}_{\{n' \in OPEN\}} \Phi(n')$

- WA* uses $\Phi(n) = f_{\text{GBFS}}(n) = h(n)$
  - Ignores the heuristic completely

- Also called **Pure Heuristic Search**

UNIVERSITY OF TORONTO

Greedy Best-First Search

- GBFS is commonly used in domain-independent planners
- Usually faster than A* and low-weight WA*
- GBFS is complete but suboptimal
  - No bound on suboptimality

Modern Optimal Search Research

- Low memory algorithms
  - IDA*, RBFS, EPEA*, SMA*, …
- Better heuristics
- Pruning methods for transpositions
  - Stubborn sets
- Bidirectional Search
  - MM, SFBDS, …

Suboptimal Search Research

- Non-uniform cost domains
  - GBFS and WA* can struggle if action costs vary greatly
- Understanding impact of different decisions
  - Re-expansions, tie-breaking, weight value
- Exploration in GBFS
  - ϵ-greedy, Type-based exploration, novelty-based pruning

UNIVERSITY OF TORONTO

Summary

- Hill-climbing as a simple way to use a heuristic
- Generalized UCS to the OCL algorithm framework
  - Showed how Best-First Search fits into this framework
- Introduced A* as an OCL algorithm
  - Considered several properties
- Considered WA* and GBFS as suboptimal alternatives