Project Part 1 Overview and Concepts

Project Part 1 requires you to complete an anytime, forward-searching, depth-bounded, utility-driven scheduler.

procedure *Search*(G, S, goal)
    Inputs
        G: graph with nodes N and arcs A
        s: start node
        goal: Boolean function of nodes
    Output
        path from s to a node for which goal is true
        or $\bot$ if there are no solution paths
    Local
        Frontier: set of paths
    Frontier $:= \{\langle s \rangle\}$ //initial Frontier to the start state
    while Frontier $\neq \{\}$ **do** //while some paths remain to be expanded
        select and **remove** $\langle n_0, \ldots, n_k \rangle$ from Frontier
        if goal (n_k) **then** //if a goal state has been reached, return solution
            return $\langle n_0, \ldots, n_k \rangle$
        Frontier $:=$ Frontier $\cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$ // generate successors
    return $\bot$

Figure 3.4: Search: generic graph searching algorithm

Lets start with a generic search algorithm

Adapted from http://artint.info/2e/html/ArtInt2e.Ch3.S4.html

Project Part 1 Overview and Concepts

Project Part 1 requires you to complete an anytime, forward-searching, depth-bounded, utility-driven scheduler.

```
procedure Search(G, S, goal)
    Inputs
        G: graph with nodes N and arcs A
        s: start node
        goal: Boolean function of nodes
    Output
        path from s to a node for which goal is true
        or ⊥ if there are no solution paths
    Local
        Frontier: set of paths
    Frontier := {⟨s⟩}
    while Frontier ≠ {} do
        select and remove ⟨n_0, ..., n_k⟩ from Frontier
        if goal(n_k) then
            return ⟨n_0, ..., n_k⟩
        Frontier := Frontier ∪ {⟨n_0, ..., n_k, n⟩ : ⟨n_k, n⟩ ∈ A}
    return ⊥
```

// if Frontier is a stack then depth-first search
// if Frontier is a queue then breadth-first search
// if Frontier is a priority queue, then some kind of "informed" search

Figure 3.4: Search: generic graph searching algorithm

Adapted from http://artint.info/2e/html/ArtInt2e.Ch3.S4.html

Project Part 1 Overview and Concepts

Project Part 1 requires you to complete an anytime, **forward-searching**, depth-bounded, utility-driven scheduler.

```
procedure Search(G, S, goal)
    Inputs
        G: graph with nodes N and arcs A
        s: start node
        goal: Boolean function of nodes
    Output
        path from s to a node for which goal is true
        or ⊥ if there are no solution paths
    Local
        Frontier: set of paths
    Frontier := {⟨s⟩}
    while Frontier ≠ {} do
        select and remove ⟨n_0, ..., n_k⟩ from Frontier
        if goal(n_k) then
            return ⟨n_0, ..., n_k⟩
        Frontier := Frontier ∪ {⟨n_0, ..., n_k, n⟩ : ⟨n_k, n⟩ ∈ A}
    return ⊥
```

Figure 3.4: Search: generic graph searching algorithm

In a forward search,

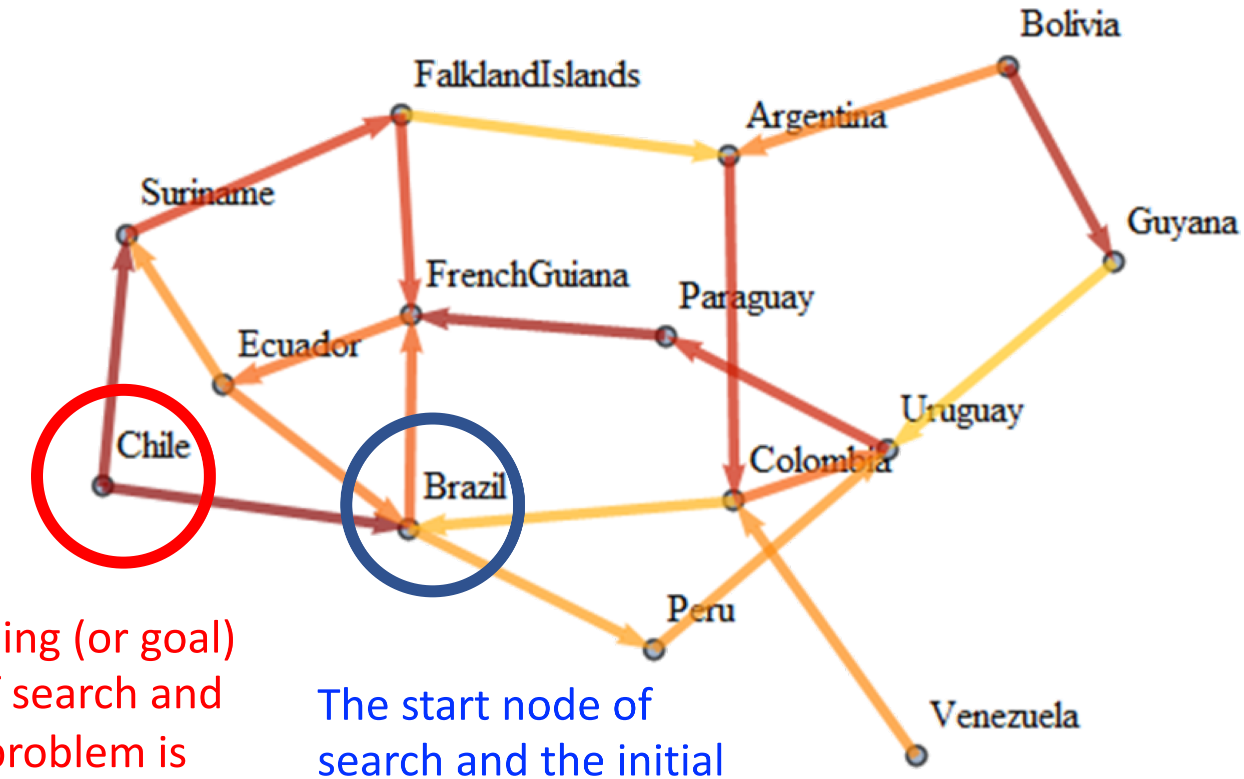
- the start node of the search is the initial state of the problem
- The goal state(s) of the search are the goal states of the problem

Adapted from http://artint.info/2e/html/ArtInt2e.Ch3.S4.html

Forward search continued

Suppose Frontier is a stack

Problem: Find a directed route from Brazil to Chili



A stopping (or goal) state of search and of the problem is “Chile”

The start node of search and the initial state of the problem is “Brazil”

In contrast, consider backward search

Suppose Frontier is a stack

Chile

Chile

Problem: Find a directed route from Brazil to Chili

The start node of search and the goal state of the problem is "Chile"

A stopping (or goal) state of search and the initial state of problem is "Brazil"

Because it's a backward search, expand the path with arcs that point INTO Chile

There are none! Search terminates with no solution exists.

In general, its often the case that backward search is faster than forward search, but your implementation should still use forward search (and one reason is that we are doing utility-driven search, not goal driven search)

Forward search continued

The previous example searched an explicit graph, but in AI (and this project) its more typical to search an implicit graph

```
procedure Search(G, S, goal)
    Inputs
        G: graph with nodes N and arcs A   // N are states and arcs can be implicit in operators
        s: start node
        goal: Boolean function of nodes
    Output
        path from s to a node for which goal is true
        or ⊥ if there are no solution paths
    Local
        Frontier: set of paths
    Frontier := {⟨s⟩}                  //initial Frontier to the start state
    while Frontier ≠ {} do             //while some paths remain to be expanded
        select and remove ⟨n0, ..., nk⟩ from Frontier
        if goal (nk) then              //if a goal state has been reached, return solution
            return ⟨n0, ..., nk⟩
        Frontier := Frontier ∪ {⟨n0, ..., nk, n⟩ : ⟨nk, n⟩ ∈ A}   // generate successors
    return ⊥
```

Figure 3.4: Search: generic graph searching algorithm

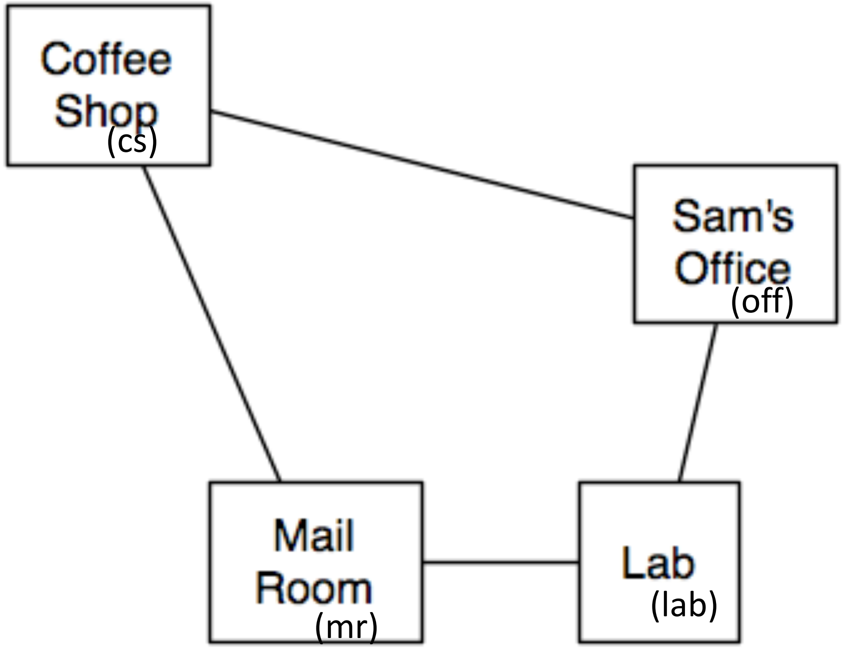
Adapted from http://artint.info/2e/html/ArtInt2e.Ch3.S4.html

Forward search continued

of an IMPLICIT graph

longish example to follow from Chapter 6 of Poole and Mackworth (http://artint.info/2e/html/ArtInt2e.Ch6.html)

Example 8 6.1 ider a delivery robot world with mail and coffee to deliver. Assume a simplified domain with four locations as shown in Figure 8 6.1 From Poole and Mackworth



Features to describe states

RLoc
– Rob's location
RHC
– Rob has coffee
SWC
– Sam wants coffee
MW
– Mail is waiting
RHM
– Rob has mail

Actions

mc
– move clockwise
mcc
– move counterclockwise
puc
– pickup coffee
dc
– deliver coffee
pum
– pickup mail
dm
– deliver mail

Explicit State-Space Representation

State	Action	Resulting State
⟨*lab*, ¬*rhc*,*swc*, ¬*mw*,*rhm*⟩	*mc*	⟨*mr*, ¬*rhc*,*swc*, ¬*mw*,*rhm*⟩
⟨*lab*, ¬*rhc*,*swc*, ¬*mw*,*rhm*⟩	*mcc*	⟨*off*, ¬*rhc*,*swc*, ¬*mw*,*rhm*⟩
⟨*off*, ¬*rhc*,*swc*, ¬*mw*,*rhm*⟩	*dm*	⟨*off*, ¬*rhc*,*swc*, ¬*mw*, ¬*rhm*⟩
⟨*off*, ¬*rhc*,*swc*, ¬*mw*,*rhm*⟩	*mcc*	⟨*cs*, ¬*rhc*,*swc*, ¬*mw*,*rhm*⟩
⟨*off*, ¬*rhc*,*swc*, ¬*mw*,*rhm*⟩	*mc*	⟨*lab*, ¬*rhc*,*swc*, ¬*mw*,*rhm*⟩
...	...	...

Initial State: {cs, ~rhc, swc, mw, ~rhm}

Goal State: {~swc}

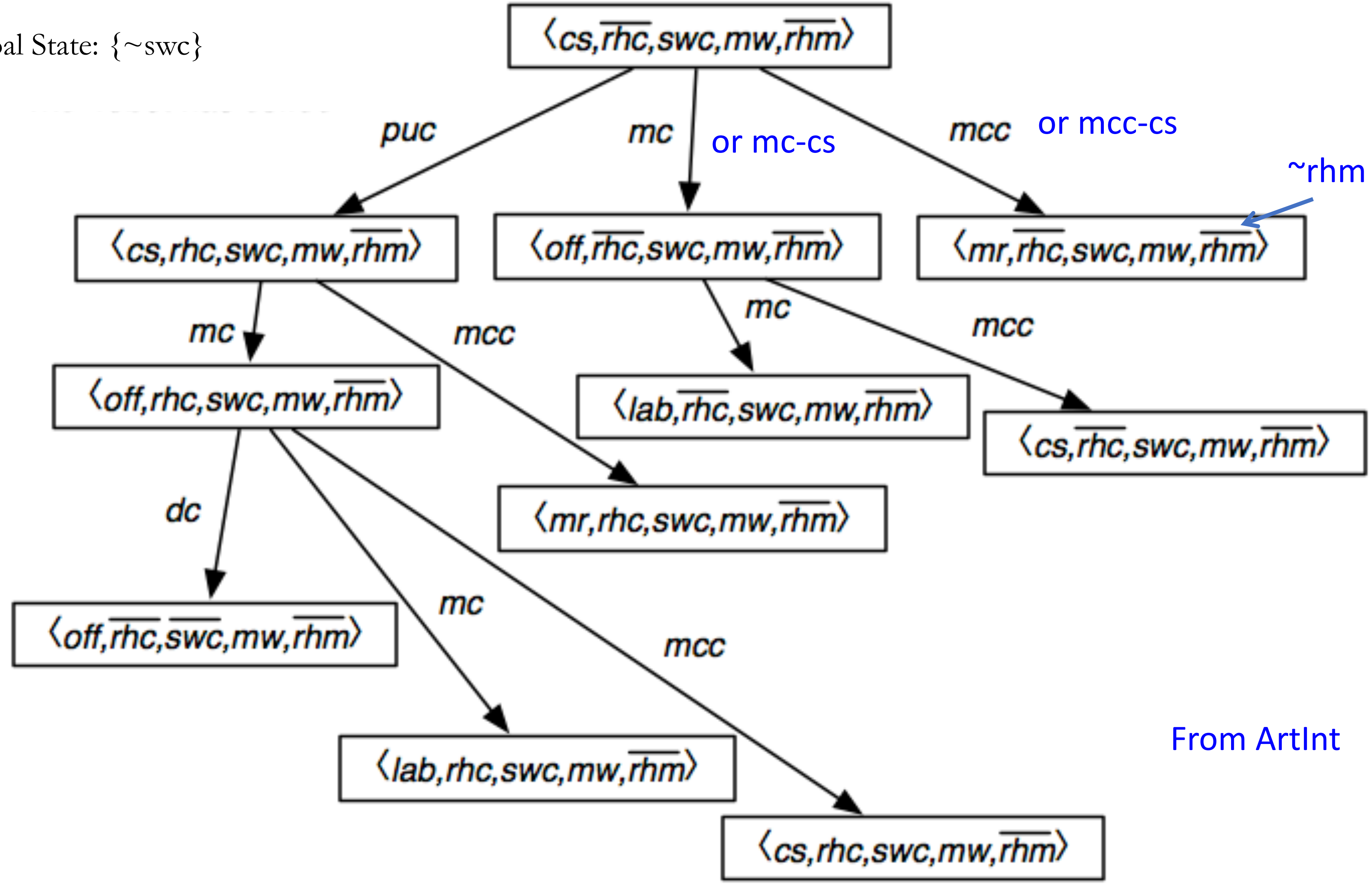


Figure 6.2 t of the search space for a state-space planner

A depth-first forward search

puc: Precondition {cs, ~rhc};
Effect {rhc}
mc-cs: Precondition {cs};
Effect {off}
dc: Precondition {off, rhc};
Effect {~rhc, ~swc}

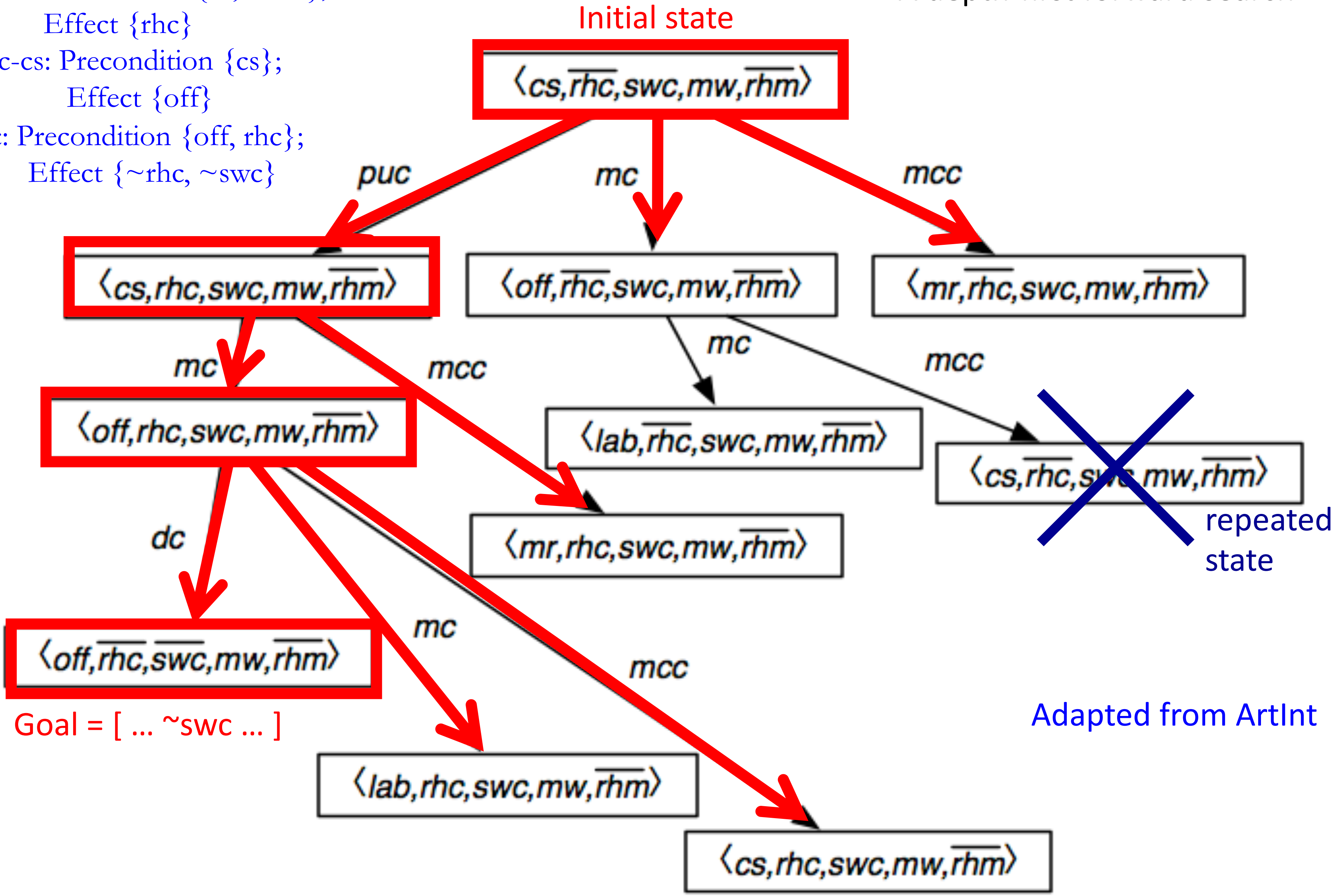


Figure 6.2 …t of the search space for a state-space planner

Thus far we just have a tabular representation of on explicit graph

Implicit arcs (i.e., operators) are used to generate resulting (or successor) states on demand

Features to describe states

RLoc
- Rob's location (4-valued)

RHC
- Rob has coffee (binary)

SWC
- Sam wants coffee (binary)

MW
- Mail is waiting (binary)

RHM
- Rob has mail (binary)

Actions

mc
- move clockwise

mcc
- move counterclockwise

puc
- pickup coffee

dc
- deliver coffee

pum
- pickup mail

dm
- deliver mail

State	Action	Resulting State
< lab, rhc, swc, mw, rhm>	mc	< mr, rhc, swc, mw, rhm>
< lab, rhc, swc, mw, ~rhm>	mc	< mr, rhc, swc, mw, ~rhm>
< lab, rhc, swc, ~mw, rhm>	mc	< mr, rhc, swc, ~mw, rhm>
< lab, rhc, swc, ~mw, ~rhm>	mc	< mr, rhc, swc, ~mw, ~rhm>
< lab, rhc, ~swc, mw, rhm>	mc	< mr, rhc, ~swc, mw, rhm>
< lab, rhc, ~swc, mw, ~rhm>	mc	< mr, rhc, ~swc, mw, ~rhm>
< lab, rhc, ~swc, ~mw, rhm>	mc	< mr, rhc, ~swc, ~mw, rhm>
< lab, rhc, ~swc, ~mw, ~rhm>	mc	< mr, rhc, ~swc, ~mw, ~rhm>
< lab, ~rhc, swc, mw, rhm>	mc	< mr, ~rhc, swc, mw, rhm>
…		
< lab, ~rhc, ~swc, ~mw, ~rhm>	mc	< mr, ~rhc, ~swc, ~mw, ~rhm>
<lab, ?V1, ?V2, ?V3, ?V4>	**mc**	**<mr, ?V1, ?V2, ?V3, ?V4>**

State	Action	Resulting State
$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$	*mc*	$\langle mr, \neg rhc, swc, \neg mw, rhm \rangle$
$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$	*mcc*	$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$
$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$	*dm*	$\langle off, \neg rhc, swc, \neg mw, \neg rhm \rangle$
$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$	*mcc*	$\langle cs, \neg rhc, swc, \neg mw, rhm \rangle$
$\langle off, \neg rhc, swc, \neg mw, rhm \rangle$	*mc*	$\langle lab, \neg rhc, swc, \neg mw, rhm \rangle$
…	…	…

Adapted from Poole and Mackworth

STRIPS Operators , which I will typically write pre(op) ➔ eff(op)

puc: {RHC = ~rhc, RLOC = cs} ➔ {RHC = rhc}

dc: {RHC = rhc, RLOC = off} ➔ {RHC = ~rhc, SWC = ~swc}

mc_cs: {RLOC = cs} ➔ {RLOC = off}

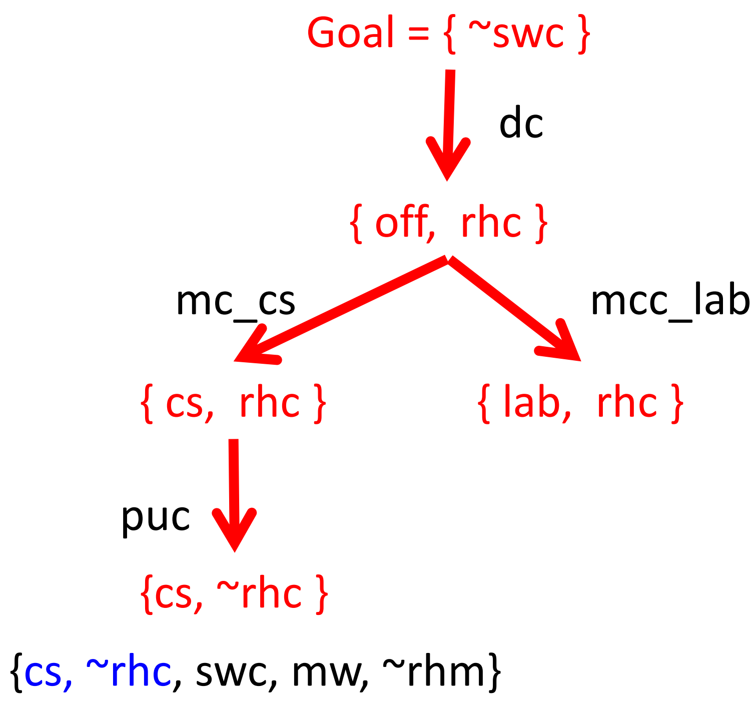
mcc_lab = {RLOC = lab} ➔ {RLOC = off}

. . .

Initial State: {cs, ~rhc, swc, mw, ~rhm}

Goal State: {~swc}

Regression or backward planning



STRIPS Operators , which I will typically write pre(op) ➔ eff(op)

puc: {RHC = ~rhc, RLOC = cs} ➔ {RHC = rhc}

dc: {RHC = rhc, RLOC = off} ➔ {RHC = ~rhc, SWC = ~swc}

mc_cs: {RLOC = cs} ➔ {RLOC = off}

mcc_off = {RLOC = off} ➔ {RLOC = cs}

. . .

Forward search in Project Part 1

procedure *Search*(G, S, goal)
    Inputs
        G: graph with nodes N and arcs A
        s: start node
        goal: Boolean function of nodes
    Output
        path from s to a node for which goal is true
        or $\bot$ if there are no solution paths
    Local
        Frontier: set of paths
    $\text{Frontier} := \{\langle s \rangle\}$
    while $\text{Frontier} \neq \{\}$ **do**
        select and **remove** $\langle n_0, \ldots, n_k \rangle$ from Frontier
        if goal (n_k) **then**
            return $\langle n_0, \ldots, n_k \rangle$
        $\text{Frontier} := \text{Frontier} \cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$ // generate successors
    return $\bot$

Figure 3.4: Search: generic graph searching algorithm

Alloys Template

((TRANSFORM ?C (INPUTS (R1 1) (R2, 2)) (OUTPUTS (R1 1) (R21, 1) (R21’ 1)),
preconditions are of the form ?ARj <= ?C(?Rj)

. . .

Electronics Template

(TRANSFORM ?C (INPUTS (R1 3) (R2 2) (R21 2)) (OUTPUTS (R22 2) (R22’ 2) (R1 3)),
preconditions are of the form ?ARj <= ?C(?Rj)

. . .

A(tlantis)	E(rewon)
R1: 500	R1: 100
R2: 700	R2: 50
R3: 100	R3: 2000
R21: 0	R21: 30
R21’: 0	R21’: 0
R22: 0	R22: 0
R22’: 0	R22’: 0
R23: 0	R23: 0
R23’: 0	R23’: 0

State, n_k

Housing Template

(TRANSFORM ?C (INPUTS (R1 5) (R2, 1) (R3 5) (R21 3) (OUTPUTS (R1 5) (R23, 1) (R23’ 1)),
preconditions are of the form ?AIk <= ?C(?Rk)

. . .

(TRANSFER ?Cj1 ?Cj2 ((?Ri ?ARi)), where ?ARi <= ?Cj1(?Ri)

. . .

Possible Pseudocode for Generate Successors

```
Successors ← { }
For each (skeletal, variablized) operator (i.e., TRANSFER and each TRANSFORM template), ?Op {
    For each variable ?X in ?Op {
        For each constant, K, of the appropriate type (i.e., country, resource, amount) {
            Substitute K for ?X in ?Op
        }
    } // when done, all variables in ?Op replaced by constants, yielding Op
    If preconditions of Op satisfied, apply Op to current world, and add successor to set of successors
}
```

How many successors (ballpark) will there be: (P ?ops) * (M vars per ?op) * (N vals per var) = P*M*N
So, in our **toy problem** of 6 countries, 9 resources, and assuming only 3 possible values per resource (lets say and average of 6 values per variable), that's

4 templates * 4 variables per template * 6 values per variable, or say 4 * 4 * 6, on the order of **100 successors**

Alloys Template

((TRANSFORM ?C (INPUTS (R1 1) (R2, 2)) (OUTPUTS (R1 1) (R21, 1) (R21' 1)),
preconditions are of the form ?ARj <= ?C(?Rj)

. . .

Electronics Template

(TRANSFORM ?C (INPUTS (R1 3) (R2 2) (R21 2)) (OUTPUTS (R22 2) (R22' 2) (R1 3)),
preconditions are of the form ?ARj <= ?C(?Rj)

. . .

Housing Template

(TRANSFORM ?C (INPUTS (R1 5) (R2, 1) (R3 5) (R21 3) (OUTPUTS (R1 5) (R23, 1) (R23' 1)),
preconditions are of the form ?AIk <= ?C(?Rk)

. . .

(TRANSFER ?Cj1 ?Cj2 ((?Ri ?ARi)), where ?ARi <= ?Cj1(?Ri)

. . .

A(tlantis)	E(rewon)
R1: 500	R1: 100
R2: 700	R2: 50
R3: 100	R3: 2000
R21: 0	R21: 30
R21': 0	R21': 0
R22: 0	R22: 0
R22': 0	R22': 0
R23: 0	R23: 0
R23': 0	R23': 0

State, n_k

A(tlantis)	E(rewon)
R1: 500	R1: 100
R2: 700	R2: 50
R3: 100	R3: 2000
R21: 0	R21: 30
R21’: 0	R21’: 0
R22: 0	R22: 0
R22’: 0	R22’: 0
R23: 0	R23: 0
R23’: 0	R23’: 0

Alloys Template

((TRANSFORM ?C (INPUTS (R1 1) (R2, 2)) (OUTPUTS (R1 1) (R21, 1) (R21’ 1)),
preconditions are of the form ?ARj <= ?C(?Rj)

(TRANSFORM **A** (INPUTS (R1 50*1) (R2, 50*2)) (OUTPUTS (R1 50) (R21, 50) (R21’ 50)),
preconditions 50 <= 500, 100 <= 700

. . .

Electronics Template

(TRANSFORM ?C (INPUTS (R1 3) (R2 2) (R21 2)) (OUTPUTS (R22 2) (R22’ 2) (R1 3)),
preconditions are of the form ?ARj <= ?C(?Rj)

(TRANSFORM **A** (INPUTS (R1 30) (R2 20) (R21 20)) (OUTPUTS (R22 20) (R22’ 20) (R1 30)),
preconditions 30 <= 500, 20 <= 700, 20 !<= 0

. . .

Housing Template

(TRANSFORM ?C (INPUTS (R1 5) (R2, 1) (R3 5) (R21 3) (OUTPUTS (R1 5) (R23, 1) (R23’ 1)),
preconditions are of the form ?AIk <= ?C(?Rk)

(TRANSFORM **E** (INPUTS (R1 10*5) (R2, 10*1) (R3 10*5) (R21 10*3) (OUTPUTS (R1 10*5) (R23, 10*1) (R23’ 10*1)),
preconditions are of the form 50 <= 100, 10 <= 50, 50 <= 2000, 30 <= 30

. . .

(TRANSFER ?Cj1 ?Cj2 ((?Ri ?ARi)), where ?ARi <= ?Cj1(?Ri)

(TRANSFER E A ((R3 500)), preconditions 500 <= 2000

. . .

A(tlantis)	E(rewon)
R1: 500	R1: 100
R2: 700	R2: 50
R3: 100	R3: 2000
R21: 0	R21: 30
R21’: 0	R21’: 0
R22: 0	R22: 0
R22’: 0	R22’: 0
R23: 0	R23: 0
R23’: 0	R23’: 0

Alloys Template

((TRANSFORM ?C (INPUTS (R1 1) (R2, 2)) (OUTPUTS (R1 1) (R21, 1) (R21’ 1)),
preconditions are of the form ?ARj <= ?C(?Rj)

(TRANSFORM **A** (INPUTS (R1 50*1) (R2, 50*2)) (OUTPUTS (R1 50) (R21, 50) (R21’ 50)),
preconditions 50 <= 500, 100 <= 700

A(tlantis)	E(rewon)
R1: 500	R1: 100
R2: 600	R2: 50
R3: 100	R3: 2000
R21: 50	R21: 30
R21’: 50	R21’: 0
R22: 0	R22: 0
R22’: 0	R22’: 0
R23: 0	R23: 0
R23’: 0	R23’: 0

. . .

Electronics Template

(TRANSFORM ?C (INPUTS (R1 3) (R2 2) (R21 2)) (OUTPUTS (R22 2) (R22’ 2) (R1 3)),
preconditions are of the form ?ARj <= ?C(?Rj)

(TRANSFORM **A** (INPUTS (R1 30) (R2 20) (R21 20)) (OUTPUTS (R22 20) (R22’ 20) (R1 30)),
preconditions 30 <= 500, 20 <= 700, 20 !<= 0

X No successor

. . .

Housing Template

(TRANSFORM ?C (INPUTS (R1 5) (R2, 1) (R3 5) (R21 3) (OUTPUTS (R1 5) (R23, 1) (R23’ 1)),
preconditions are of the form ?Alk <= ?C(?Rk)

(TRANSFORM **E** (INPUTS (R1 10*5) (R2, 10*1) (R3 10*5) (R21 10*3) (OUTPUTS (R1 10*5)
(R23, 10*1) (R23’ 10*1)),
preconditions are of the form 50 <= 100, 10 <= 50, 50 <= 2000, 30 <= 30

A(tlantis)	E(rewon)
R1: 500	R1: 10
R2: 700	R2: 40
R3: 100	R3: 1950
R21: 0	R21: 0
R21’: 0	R21’: 0
R22: 0	R22: 0
R22’: 0	R22’: 0
R23: 0	R23: 10
R23’: 0	R23’: 10

. . .

(TRANSFER ?Cj1 ?Cj2 ((?Ri ?ARi)), where ?ARi <= ?Cj1(?Ri)

(TRANSFER E A ((R3 500)), preconditions 500 <= 2000

A(tlantis)	E(rewon)
R1: 500	R1: 100
R2: 700	R2: 50
R3: 600	R3: 1500
R21: 0	R21: 30
R21’: 0	R21’: 0
R22: 0	R22: 0
R22’: 0	R22’: 0
R23: 0	R23: 0
R23’: 0	R23’: 0

. . .

This shows only a few of the many successors in our domain

Other thoughts

- The nested-loops pseudocode I outline might be made more efficient by checking preconditions earlier
- Generate successors is needed for any of the AI search variants you might use; the function is not mentioned by name using the generic search algorithm found in Poole and Mackworth, but it is implicit in line 16 of figure 3.4 where they reference a (generated) set that is unioned with the frontier (https://artint.info/2e/html/ArtInt2e.Ch3.S4.html). In Russell and Norvig, Section 3.3 (and Figure 3.7) they refer to this as generating or expanding nodes.
- ASIDE: Generate successor states of a node all at once as specified, but an alternative (and one that Russell and Norvig refers to, albeit inconsistently) is rather than generating the successor states all at once, form pairs of form (current state, Op), where Op is a grounded (constants only Op), and apply the Op to the current state to get a successor state "as needed" . This can be more efficient. Note this (e.g., for the next quiz), but don't implement it it for the pre-break deliverable.
- There are still issues/ambiguities that you must address
- More generally, **you will be faced with issues about the spec that you will have to decide upon**. For example, in generating successors, you might decide that generating successors for every possible integer value of various resource amounts, and combinations thereof, might be way too expensive, and you might consider binning the value domains of each country's resources (e.g., 10%, 25%, 50%, 100% would be four bins). or using

Project Part 1 requires you to complete an **anytime**, forward-searching, depth-bounded, utility-driven scheduler.

procedure *Search*(G, S, goal)
    Inputs
        G: graph with nodes N and arcs A
        s: start node
        goal: Boolean function of nodes
    Output
        path from s to a node for which goal is true
        or $\bot$ if there are no solution paths
    Local
        Frontier: set of paths
        ; Solutions: Priority Queue of solutions organized by solution "quality"
    Frontier := $\{\langle s \rangle\}$
    ; Solutions := Empty Priority Queue
    while Frontier $\neq \{\}$ **do**
        select and **remove** $\langle n_0, \ldots, n_k \rangle$ from Frontier
        if goal (n_k) **then**
            ~~**return** $\langle n_0, \ldots, n_k \rangle$~~
            ; add <n0,…,nk> to Solutions using solution "quality"
        else Frontier := Frontier ∪
    return $\bot$

Figure 3.4: Search: generic graph searching algorithm

This is changed from a termination step, to a step that adds the solution to a set of solutions and continues searching

Adapted from http://artint.info/2e/html/ArtInt2e.Ch3.S4.html

Project Part 1 requires you to complete an anytime, forward-searching, **depth-bounded**, **utility-driven** scheduler.

procedure *Search*($G, S,$ ~~goal~~) D: depth bound

Inputs

G: graph with nodes N and arcs A

s: start node U: utility function (applied to a path, not a single node)

goal: Boolean function of nodes

Output

path from s to a node for which goal is true

or $\bot$ if there are no solution paths

Local

Frontier: set of paths ; Solutions: Priority Queue of solutions organized by solution "quality", presumably by U

Frontier := $\{\langle s \rangle\}$; Solutions := Empty Priority Queue

while Frontier $\neq \{\}$ **do**

select and **remove** $\langle n_0, \ldots, n_k \rangle$ from Frontier

if depth(n_k) >= D then

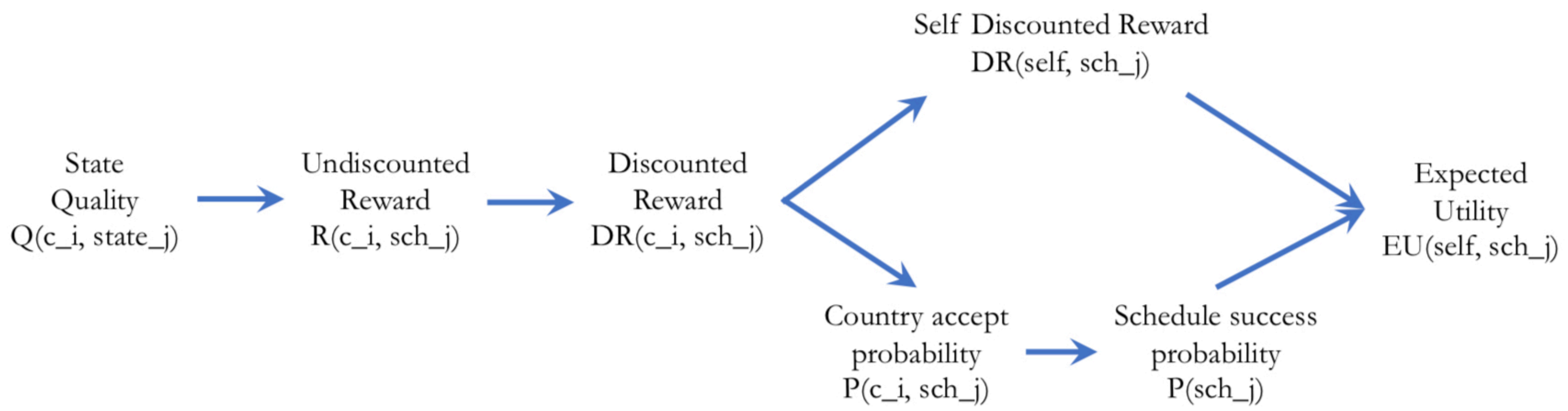
~~**return** $\langle n_0, \ldots, n_k \rangle$~~ ; add <n0,…,nk> to Solutions using solution "quality", presumably by U

else Frontier := Frontier $\cup \{\langle n_0, \ldots, n_k, n \rangle : \langle n_k, n \rangle \in A\}$

return $\bot$

Figure 3.4: Search: generic graph searching algorithm

Adapted from http://artint.info/2e/html/ArtInt2e.Ch3.S4.html


State
Quality
Q(c_i, state_j)
Undiscounted
Reward
R(c_i, sch_j)
Discounted
Reward
DR(c_i, sch_j)
Self Discounted Reward
DR(self, sch_j)
Country accept
probability
P(c_i, sch_j)
Schedule success
probability
P(sch_j)
Expected
Utility
EU(self, sch_j)