Constructive Artificial Intelligence

Search

Daniel Polani

School of Computer Science
University of Hertfordshire

October 30, 2009

All rights reserved. Permission is granted to copy and distribute these slides in full or in part for purposes of research, education as well as private use, provided that author, affiliation and this notice is retained.
Use as part of home- and coursework is only allowed with express permission by the responsible tutor and, in this case, is to be appropriately referenced.

Search

In a search, one begins with a known configuration and attempts to find the path towards a desired target configuration according to allowed transitions. More precisely:

Definition (Search Problem)

A **search problem** is defined by its

State/Node Set: a set $\mathcal{N}$ of **nodes** (also called **states**), each of which describes a particular situation that could (theoretically) occur in the given scenario;

Successor Rule: for each node $n \in \mathcal{N}$, the function S generating the set $S(n) \subseteq \mathcal{N}$ of successors of n;

Start Node: one node s, at which the search starts;

Goal State Set: a set $\mathcal{G} \subseteq \mathcal{N}$ of one or more goal nodes of the search.

Goal

Find a **path** from $n_0 := s$ to some $n_k := g \in \mathcal{G}$, i.e. a sequence $n_0, n_1, n_2, \ldots, n_k$ such that n_{i+1} is a successor of n_i.

Search Interface

Example (search.py — Interface Construction in Python)

```
####################################################
#
# this is just an interface, do not use on its own
#
####################################################

class Nodes:
    def succ(self, n):
        raise Exception, "Successor undefined"
    def start(self):
        raise Exception, "Start undefined"
    def goal(self, n):
        raise Exception, "Goal undefined"
```

Exercise 1 (Wolf/Cabbage/Goat Problem)

Exercise

Formulate the Wolf/Cabbage/Goat Problem in terms of state variables, start node, goal node and transitions and write a class that represents it.

Wolf/Cabbage/Goat I

```
##########################################################################
# wolf_cabbage_goat.py
##########################################################################

import search                           # get the interface

def other_side(left_right):
    if left_right == 'left': return 'right'
    elif left_right == 'right': return 'left'

def safe(node):
    # the side without the farmer needs check,
    # the other is safe
    side_without_farmer = other_side(node['farmer'])

    lone_travelers = set(traveler for traveler in node
                         if node[traveler] == side_without_farmer)

    # dangeroos animals
    unsafe = set(['wolf', 'goat']).issubset(lone_travelers) or\
             set(['goat', 'cabbage']).issubset(lone_travelers)
    return not unsafe

class Wolf_Cabbage_Goat(search.Nodes):
    def start(self):
        # farmer, wolf, cabbage, goat on one side
        return { 'farmer': 'left',
                 'wolf': 'left',
                 'goat': 'left',
                 'cabbage': 'left' }
```

Wolf/Cabbage/Goat II

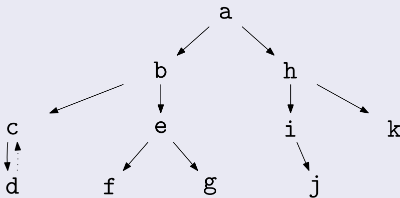
```
def goal(self, node):
    # true if all moved to right
    return set(node[i] for i in node) == set(['right'])

def succ(self, node):
    for traveler in node.keys() + ['']:
        # for all possible travelers, including none

        if traveler == 'farmer':
            continue
        # ignore, we'll deal separately with him

        new_node = node.copy()
        # create a safety copy

        # if somebody wants to travel, and the farmer is on the
        # same side, do it

        if traveler and new_node[traveler] == new_node['farmer']:
            new_node[traveler] = other_side(new_node[traveler])

        # the farmer always travels
        new_node['farmer'] = other_side(new_node['farmer'])

        # after a trip, the new state is ok if no one gets eaten
        if safe(new_node):
            yield new_node
```

Depth-First Search

For a given node n, search solution path $\mathrm{Sol}(n)$ to some goal node, by considering

Base Case: if $n \in \mathcal{G}$, define the solution path $\mathrm{Sol}(n)$ by $[n]$;

Recursion Step: else if n has a successor n' with a solution path $\mathrm{Sol}(n') = [n', \ldots, g]$ (reduced problem), then define solution path $\mathrm{Sol}(n) := [n] + \mathrm{Sol}(n') = [n, n', \ldots, g]$



Depth-First Search

Example (depth_first_search.py — DFS-Algorithm)

```
def depth_first_search(problem, node):
    if problem.goal(node): return [node]
    # base case

    for n_succ in problem.succ(node):
        sol = depth_first_search(problem, n_succ)
        if sol:
            # first path is returned
            return [node] + sol
```

Example (wcg_run.py — Running Code)

```
import wolf_cabbage_goat
from depth_first_search import *

wcg = wolf_cabbage_goat.Wolf_Cabbage_Goat()
print depth_first_search(wcg, wcg.start())
```

Problems of Depth-First Search

Observations

- parsimonious storage of candidate solution paths
- if the goals happen to be in the wrong branch, the search can take long

Problems of Depth-First Search

Observations

- parsimonious storage of candidate solution paths
- if the goals happen to be in the wrong branch, the search can take long
- naive Depth-First can fail if the search tree is infinitely deep at places or contains repeats (loops)

Problems of Depth-First Search

Observations

- parsimonious storage of candidate solution paths
- if the goals happen to be in the wrong branch, the search can take long
- naive Depth-First can fail if the search tree is infinitely deep at places or contains repeats (loops)
- Protection against repeats: record all past positions

Depth-First Search with State Records I

Example (wcg_token.py — Modified Problem)

```
import wolf_cabbage_goat

class WCG_Token(wolf_cabbage_goat.Wolf_Cabbage_Goat):
    def token(self, node):
        # get the list of pairings of traveler and location
        # each of them e.g. ('goat', 'right')
        pairs = list(node.iteritems())

        # sort them (arbitrarily, but fixed), so that
        # the token does not depend on arbitrary
        # dictionary order
        pairs.sort()

        # make it immutable, turn it into tuple
        return tuple(pairs)
```

Depth-First Search with State Records II

Example (depth_first_search_hash.py — Modified DFS-Search)

```
def depth_first_search(problem, node, visited = set()):
    token = problem.token(node)
    if token in visited: return
    # we have tried this node already

    visited = visited.union(set([token]))
    # create new set that contains the node

    if problem.goal(node): return [node]
    # base case

    for n_succ in problem.succ(node):
        sol = depth_first_search(problem, n_succ, visited)
        if sol:
            # first path is returned
            return [node] + sol
```

Example (wcg_run_token.py — Modified Running Code)

```
import wcg_token
import depth_first_search_hash

wcg = wcg_token.WCG_Token()

print depth_first_search_hash.depth_first_search(wcg, wcg.start())
```