

# Constructive Artificial Intelligence

## Search II

Daniel Polani

School of Computer Science  
University of Hertfordshire

November 10, 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.

# Exercise 1 (6 Coins Problem)

## Exercise

Formulate the 6-Coins Problem in terms of state variables, start node, goal node and transitions and write a class that represents it.

## Breadth-First Search

- 1 For a given node  $n$ , search solution path  $\text{Sol}(n)$  to some goal node.

For this, consider the **breadth-first search** on the candidate list  $\mathcal{C}$  of partial paths for the solution path, starting with  $\mathcal{C} := \{[n]\}$ .

- 2 Pop (pick and remove) first partial path  $[n, \dots, n']$  from candidate list  $\mathcal{C}$ . If none, **failure**. Else:

**Base Case:** if  $n' \in \mathcal{G}$ , define solution path  $\text{Sol}(n)$  by  $[n, \dots, n']$ ;

**Recursion Step:** if not,  $\text{Sol}(\mathcal{C})$  is given by the

- 1 generating all 1-step extensions  $[n, \dots, n', n'']$  where  $n''$  is a successor of  $n'$
- 2 appending them to candidate list  $\mathcal{C}$ , resulting in a new candidate list  $\mathcal{C}'$ ;
- 3 and calculation of  $\text{Sol}(\mathcal{C}')$ .

# Breadth-First Search (Code) I

## Breadth-First Search Function

```
def breadth_first_search(problem, candidates):
    if not candidates: return
    # make sure there is something in the candidate list

    # I am modifying 'candidates' list here.
    # Why don't I need to copy?

    c = candidates.pop(0)    # pop from front
    node = c[-1]             # must exist

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

    succ = [s for s in problem.succ(node)]

    for s in problem.succ(node):
        candidates.append(c + [s])
        # 1-step extension

    return breadth_first_search(problem, candidates)
```

# Breadth-First Search (Code) II

## Calling Code

```
import wolf_cabbage_goat
from breadth_first_search import *

wcg = wolf_cabbage_goat.Wolf_Cabbage_Goat()
print breadth_first_search(wcg, [[wcg.start()]])

# careful how to specify the initial candidates:
# it's not a node, but a candidate list of paths,
# i.e. a list of lists
```

# Extension to Breadth-First Search

## Note

- Best-First Search:
- breadth-first selects the shortest path
  - best-first selects the least “costly” path

For that: define  $c(n, n')$  as cost for moving from node  $n$  to  $n'$

## Idea

- Concept:
- 1 right now, paths in candidate list sorted by ascending length
  - 2 how about sorting them by ascending cost?

Questions:

- 1 how to keep list sorted?

# Extension to Breadth-First Search

## Note

- Best-First Search:
- breadth-first selects the shortest path
  - best-first selects the least “costly” path

For that: define  $c(n, n')$  as cost for moving from node  $n$  to  $n'$

## Idea

- Concept:
- 1 right now, paths in candidate list sorted by ascending length
  - 2 how about sorting them by ascending cost?

- Questions:
- 1 how to keep list sorted? **Heaps! — check out the [heapq module in Python](#)**
  - 2 we only know past candidate length/cost: how to estimate the expected total cost?

# Best-First Search I

## Best-First Search Code

```
import heapq
import path

# candidates is a list of Path-s

def best_first_search(problem, candidates):
    if not candidates: return
    # make sure there is something in the candidate list

    #print "Candidates", candidates

    cand = heapq.heappop(candidates) # pop best candidate path
    node = cand.current()           # current node must exist

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

    for succ_node in problem.succ(node):
        heapq.heappush(candidates, cand.appended(succ_node))
        # 1-step extension

    return best_first_search(problem, candidates)
```

# Best-First Search II

## Calling Code

```
import wolf_cabbage_goat
from best_first_search import *
import wcg_path

wcg = wolf_cabbage_goat.Wolf_Cabbage_Goat()
start_path = wcg_path.WCG_Path(path = [wcg.start()], length = 1)
print best_first_search(wcg, [start_path])
```

# Implementing Path Costs — Base Path Class (path.py)

```
class Path:
    def __init__(self, path = [], length = 0):
        # current path, current length (cost)
        self.path = path
        self.length = length

    def __le__(self, path2):
        # default comparison by cost
        return self.length <= path2.length

    def __len__(self): return self.length

    def cost(self, n_from, n_to): return 1 # default cost

    def current(self):
        # default: last node in path
        if not self.path:
            raise Exception, "Path is empty"
        return self.path[-1]

    def appended(self, node):
        # Return path with appended node (and record new length).
        # This is so general that it can be in Path

        # careful to return an object of the original class by which
        # we were called
        return self.__class__(path = self.path + [node],
                               length = self.length + self.cost(self.current(), node))
```

# Specialization Class (wcg\_path.py)

```
from path import *

class WCG_Path(Path):
    def graph(self, n):
        return " ".join(p for p in n if n[p] == 'left') + "----" + \
            " ".join(p for p in n if n[p] == 'right')

    def __repr__(self):
        return "\n".join(self.graph(n) for n in self.path)
```



Judea Pearl.

*Heuristics: Intelligent Search Strategies for Computer Problem-Solving.*

Addison Wesley, 1984.



S. Russell and P. Norvig.

*Artificial Intelligence: A Modern Approach.*

Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2  
edition, 2002.