Automated Planning

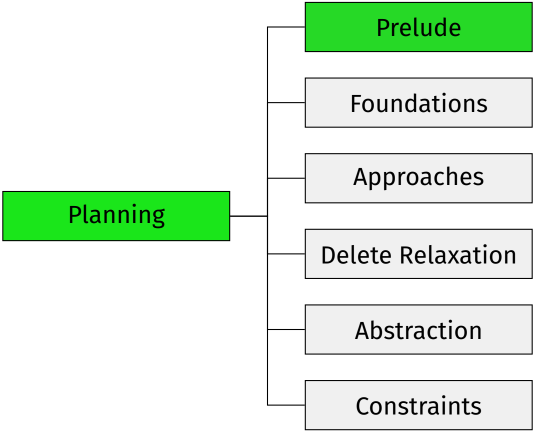
A3. Getting to Know a Planner

Jendrik Seipp

Linköping University

based on slides from the AI group at the University of Basel

Content of this Course



PDDL

PDDL

- short for Planning Domain Definition Language
- widely used in the planning community
- modelling language to describe planning domains
- separates domain description from instance description

PDDL

- short for Planning Domain Definition Language
- widely used in the planning community
- modelling language to describe planning domains
- separates domain description from instance description

course relevance: only for labs and examples

History of PDDL

More and more expressive versions have been published:

- 1998: PDDL 1.2 (basic version)
- 2002: PDDL 2.1 (numeric and temporal features)
- 2004: PDDL 2.2 (derived predicates and timed initial literals)
- 2004: PPDDL (probabilistic)
- 2006: PDDL 3 (soft goals and trajectory constraints)
- 2006: PDDL+ (continuous state spaces)

History of PDDL

More and more expressive versions have been published:

- 1998: PDDL 1.2 (basic version)
- 2002: PDDL 2.1 (numeric and temporal features)
- 2004: PDDL 2.2 (derived predicates and timed initial literals)
- 2004: PPDDL (probabilistic)
- 2006: PDDL 3 (soft goals and trajectory constraints)
- 2006: PDDL+ (continuous state spaces)

We only consider a subset of PDDL 1.2!

Components of a PDDL planning task

- Objects that exist in the task
- Predicates that describe properties of and relations between objects
- Action schemas that describe how the current state of objects can be changed
- An initial state and a goal that describe initial and desired properties of objects

Example: The Seven Bridges of Königsberg

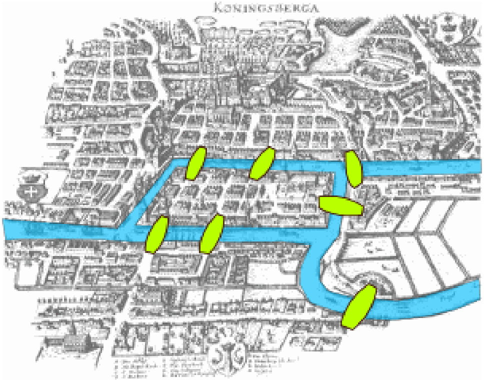


image credits: Bogdan Giuşcă (public domain)

Demo

```
$ ls demo/koenigsberg
```

PDDL Skeleton

Domain File

```
(define (domain <domain name>)
    (:requirements :strips)
    (:predicates
        <list of predicate schemata>
    )
    <list of action schemata>
)
```

PDDL Skeleton

Action Schema Specification

```
(:action <action name>
    :parameters (<list of parameters>)
    :precondition (<precondition description>)
    :effect (<effect description>)
)
```

PDDL Skeleton

Instance File

```
(define (problem <problem name>)
    (:domain <domain name>)
    (:objects
        <list of objects>
    )
    (:init
        <predicates that hold in initial state>
    )
    (:goal
        <goal description>
    )
)
```

PDDL Skeleton with action costs

Domain File

```
(define (domain <domain name>)
    (:requirements :strips :action-costs)
    (:predicates
        <list of predicate schemata>
    )
    (:functions (total-cost) - number)
    <list of action schemata>
)
```

PDDL Skeleton with action costs

Action Schema Specification

```
(:action <action name>
    :parameters (<list of parameters>)
    :precondition (<precondition description>)
    :effect (and <effect description>
                 (increase (total-cost) <action cost>)
            )
)
```

PDDL Skeleton with action costs

Instance File

```
(define (problem <problem name>)
    (:domain <domain name>)
    (:objects
        <list of objects>
    )
    (:init
        <predicates that hold in initial state>
        (= (total-cost) 0)
    )
    (:goal
        <goal description>
    )
    (:metric minimize (total-cost))
)
```

Fast Downward and VAL

Getting to Know a Planner

We now play around a bit with a planner and its input:

- look at problem formulation
- run a planner (= planning system/planning algorithm)
- validate plans found by the planner

Planner: Fast Downward

Fast Downward

We use the Fast Downward planner in this course

- because we know it well (developed by Basel AI group)
- because it implements many search algorithms and heuristics
- because it is the classical planner most commonly used as a basis for other planners

$\rightsquigarrow$ `https://www.fast-downward.org`

Validator: VAL

VAL

We use the VAL plan validation tool (Fox, Howey & Long)
to independently verify that the plans we generate are correct.

- very useful debugging tool
- `https://github.com/KCL-Planning/VAL`

15-Puzzle

Illustrating Example: 15-Puzzle

9	2	12	7
5	6	14	13
3		11	1
15	4	10	8

$\longrightarrow$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Solving the 15-Puzzle

Demo

```
$ cd demo
$ less tile/puzzle.pddl
$ less tile/puzzle01.pddl
$ ./fast-downward.py \
      tile/puzzle.pddl tile/puzzle01.pddl \
      -heuristic "h=ff()" \
      -search "eager_greedy([h],preferred=[h])"
...
$ validate tile/puzzle.pddl tile/puzzle01.pddl \
      sas_plan
...
```

Variation: Weighted 15-Puzzle

Weighted 15-Puzzle:

- moving different tiles has different cost
- cost of moving tile *x* = number of prime factors of *x*

Demo

```
$ cd demo
$ meld tile/puzzle.pddl tile/weight.pddl
$ meld tile/puzzle01.pddl tile/weight01.pddl
$ ./fast-downward.py \
      tile/weight.pddl tile/weight01.pddl \
      -heuristic "h=ff()" \
      -search "eager_greedy([h],preferred=[h])"
...
```

Variation: Glued 15-Puzzle

Glued 15-Puzzle:

- some tiles are glued in place and cannot be moved

Demo

```
$ cd demo
$ meld tile/puzzle.pddl tile/glued.pddl
$ meld tile/puzzle01.pddl tile/glued01.pddl
$ ./fast-downward.py \
      tile/glued.pddl tile/glued01.pddl \
      -heuristic "h=cg()" \
      -search "eager_greedy([h],preferred=[h])"
...
```

Note: different heuristic used!

Variation: Cheating 15-Puzzle

Cheating 15-Puzzle:

- Can remove tiles from puzzle frame (creating more blanks) and reinsert tiles at any blank location.

Demo

```
$ cd demo
$ meld tile/puzzle.pddl tile/cheat.pddl
$ meld tile/puzzle01.pddl tile/cheat01.pddl
$ ./fast-downward.py \
    tile/cheat.pddl tile/cheat01.pddl \
    -heuristic "h=ff()" \
    -search "eager_greedy([h],preferred=[h])"
...
```

Reminder: Heuristics and A* Search

Heuristics

Definition (heuristic)

Let $\mathcal{S}$ be a state space with set of states S.

A heuristic function or heuristic for $\mathcal{S}$ is a function

$$h : S \longrightarrow \mathbb{R}_0^+ \cup \{\infty\},$$

mapping each state to a non-negative number (or ∞).

Heuristics: Intuition

idea: $h(s)$ estimates cost of cheapest path
from s to closest goal state

- heuristics can be arbitrary functions
- intuition: the closer h is to true cost to goal, the more efficient the search using h

A* Search Algorithm

A* search algorithm

- based on heuristic h, define evaluation function f for node n:
  $f(n) := g(n) + h(n.\text{state})$
- trade-off between path cost and estimated proximity to goal
- intuition: $f(n)$ estimates costs of cheapest solution from initial state through n.state to goal

A* Search Algorithm: Pseudo-Code

A* search algorithm (with re-opening)

```
open := new priority queue, ordered by ⟨f, h⟩
if h(init-state) < ∞:
    open.insert(make_root_node())
distances := new HashTable
while not open.empty():
    n = open.pop-min()
    if (not distances.contains(n.state)) or (g(n) < distances[n.state]):
        distances[n.state] := g(n)
        if is-goal(n.state):
            return extract-solution(n)
        for each successor ⟨a, s′⟩ of n.state:
            if h(s′) < ∞:
                n′ := make_node(n, a, s′)
                open.insert(n′)
return unsolvable
```

A* Search Algorithm

Most important property

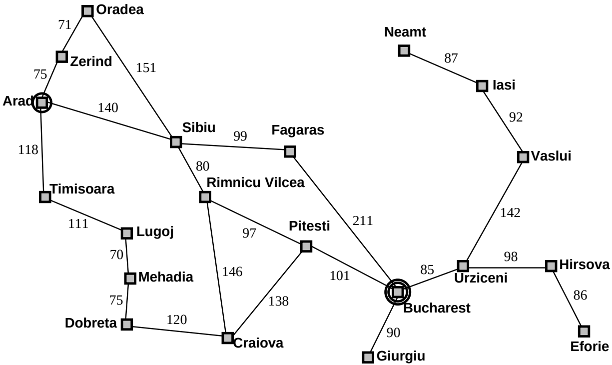
- A* is optimal if the applied heuristic is admissible.

For more details on best-first search and A*, see the TDDC17 course.

(`https://www.ida.liu.se/~TDDC17/`)

Example: A* for Route Planning

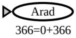
Example heuristic: straight-line distance to Bucharest



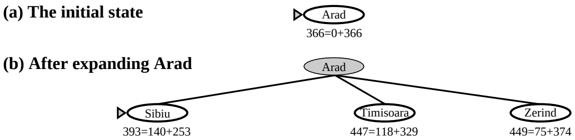
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example: A* for Route Planning

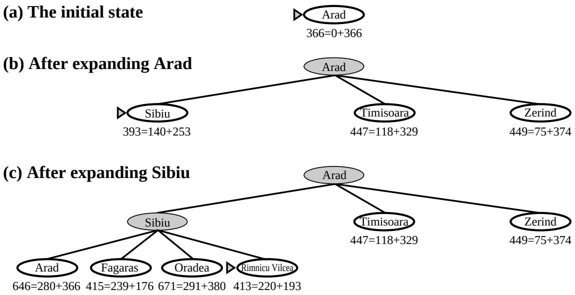
(a) The initial state



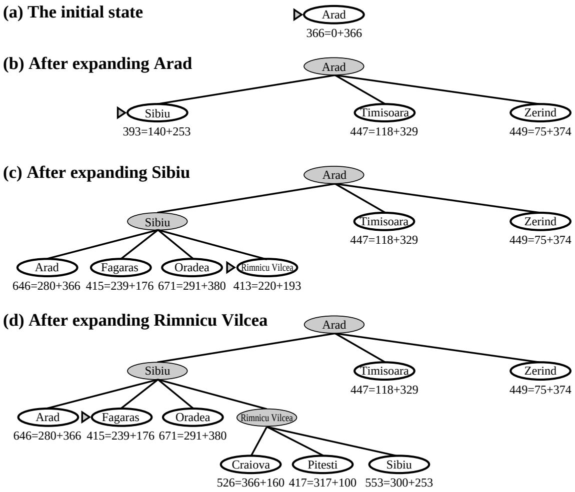
Example: A* for Route Planning



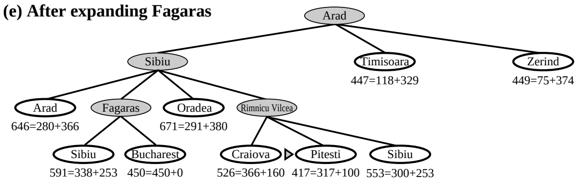
Example: A* for Route Planning



Example: A* for Route Planning

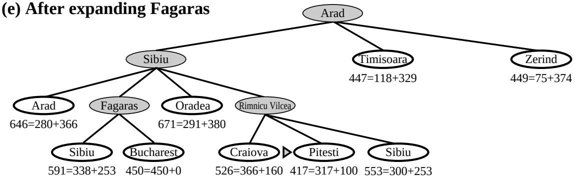


Example: A* for Route Planning

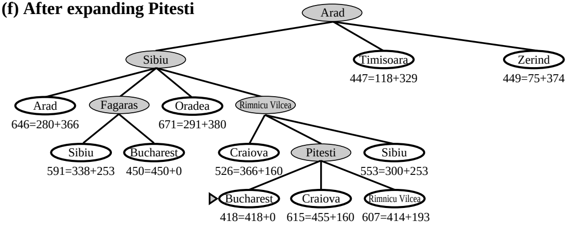


Example: A* for Route Planning

(e) After expanding Fagaras



(f) After expanding Pitesti



Solving the 15-Puzzle Optimally

Demo

```
$ cd demo
$ less tile/puzzle.pddl
$ less tile/puzzle01.pddl
$ ./fast-downward.py \
      tile/puzzle.pddl tile/puzzle01.pddl \
      -heuristic "h=lmcut()" \
      -search "astar(h)"
...
$ validate tile/puzzle.pddl tile/puzzle01.pddl \
      sas_plan
...
```

Summary

Summary

- We saw planning tasks modeled in the PDDL language.
- We ran the Fast Downward planner and VAL plan validator.
- We made some modifications to PDDL problem formulations and checked the impact on the planner.