Advanced Problem Solving Techniques
Final Project

Ilia Kurenkov
Lisa Raithel

June 10, 2017

University of Potsdam
MSc. Cognitive Systems

Advanced Problem Solving Techniques
Final Project

Ilia Kurenkov
Lisa Raithel
June 10, 2017
University of Potsdam
MSc. Cognitive Systems

Implementation

Implementation

Implementation – Intuition

- build globally optimal plan
- elevators move according to plan

Implementation – Intuition

- build globally optimal plan
- elevators move according to plan

- not really what we were supposed to do
- not possible to add new floors while running
- ...

Implementation – Idea I

- naive solution: graph navigation problem
- way too slow
- couldn't handle more than five requests and one elevator $\rightarrow$ too complex

Implementation – Idea I

- naive solution: graph navigation problem
- way too slow
- couldn't handle more than five requests and one elevator $\rightarrow$ too complex

- treated problem as graph navigation problem
- discovered this approach is way too slow

Implementation – Idea II

- derive solution incrementally in the style of a finite state machine
- fast, but failed for some instances
- difficult to understand and debug

Implementation – Idea II
- derive solution incrementally in the style of a finite state machine
- fast, but failed for some instances
- difficult to understand and debug

- coolest idea
- treat elevators and their “ways” as a finite state machine
- elevator goes from state to state, like “serving”, “moving to target floor”, “being done”
- fastest (?) solution so far, but failures
- with more complex instances we got into problems, hard to debug what caused the failures

Implementation – Helpful Clues

- care about floors, not only requests
  - floors that need to be served
  - floors with nothing to do
- some requests have to be served by specific elevators → less choice
- divide requests equally amongst elevators
- compute travel distance without recursively constructing paths through a graph
- specifying of plan in the first shot iteration

Implementation – Helpful Clues
- care about floors, not only requests
  - floors that need to be served
  - floors with nothing to do
- some requests have to be served by specific elevators → less choice
- divide requests equally amongst elevators
- compute travel distance without recursively constructing paths through a graph
- specifying of plan in the first shot iteration

- some clues we used: the actual floor position, not only requests
- distinguish about target floors and non-target floors
- deliver requests have to be served by a specific elevator → elevators have to go at least in this direction and will likely pass other floors on their way
- so save time, that is, steps, divide requests equally
- calculate travel distances to keep track of time/steps
- in the end: specified more and more of the “travel plan” before the elevators actually moved
- **venn diagram of an example 3 elevators and how requests would be divided amongst them?**

Implementation – Idea III

- identify floors that need to be served
- add deliver requests to corresponding elevator
- distribute remaining request equally
  - no elevator should be allowed to idle while another is still working
  - `move` as small a distance as possible
- behave differently when there's only one elevator

Implementation – Idea III
- identify floors that need to be served
- add deliver requests to corresponding elevator
- distribute remaining request equally
  - no elevator should be allowed to idle while another is still working
  - move as small a distance as possible
- behave differently when there's only one elevator

- wanted to have target floors and ignore the rest (only operation in remaining floors: `move`)
- deliver requests had assigned elevator, so we added them to the elevators "target list"
- wanted to assign the remaining `calls` equally
- elevators should move as little as possible to save time
- somewhat different behavior for single elevators since they have to serve all floors

Implementation – #program base.

- check if there is a request right on the floor where the elevator starts
- predicate that assigns "coordinates" from elevator to target
- get the distance to target furthest along some direction

```
% Create coordinates from elevator to target
target_coord(E, -1, F, | DIST |) :-
    init(at(elevator(E), FE)),
    target(E, F),
    DIST = FE - F,
    DIST > 0.
```


Implementation – #program base.

- check if there is a request right on the floor where the elevator starts
- predicate that assigns "coordinates" from elevator to target
- get the distance to target furthest along some direction

% Create coordinates from elevator to target
target_coord(E, -1, F, | DIST |) :-
init(at(elevator(E), FE)),
target(E, F),
DIST = FE - F,
DIST > 0.

- make sure that requests on the starting floor are served
- designed predicate that assigns the "way" from current elevator position to target position (direction in which to go, distance to travel)
- calculated distance to the targets with the greatest distance → after traveling this distance, the elevator doesn't have to go further (done with all requests)

Implementation – #program base.

- if elevator has targets above and below, the starting direction is direction of nearest target
- count `move` and `serve` operations
- combine number of `move` and `serve` operations to get the number of steps

```
% Combine serves and moves to get number of steps,
% which we then try to minimize.
n_steps(E, STEPS) :-
    STEPS = ST + D,
    total_n_serves(E, ST),
    total_n_moves(E, D),
    agent(elevator(E)).
```


Implementation – #program base.
- if elevator has targets above and below, the starting direction is direction of nearest target
- count move and serve operations
- combine number of move and serve operations to get the number of steps
% Combine serves and moves to get number of steps,
% which we then try to minimize.
n_steps(E, STEPS) :-
STEPS = ST + D,
total_n_serves(E, ST),
total_n_moves(E, D),
agent(elevator(E)).

- determine the direction in which an elevator moves first → depends on distance to first target
- count how many `moves` in total occur on each side (direction) of the elevator's starting position
- if elevator needs to change directions (targets on both sides of starting position), the distance that is traveled twice should be the smaller distance
- same goes for the number of `serves`
- combine those numbers to get the number of need steps to complete a single elevator's mission
- this is also an aspect we want to minimize later

Implementation – #program base.

Elevator Instructions:

- fix the serving times: determine *where* and *when* the elevator has to serve
- determine when elevators have to switch direction
- determine the current direction for each time step

```
% For bidirectional elevators we need to know
% when they switch direction
switch_point(E, STEP) :-
    bidirectional(E),
    initial_direction(E, DIR),
    furthest_along(E, DIR, DIST),
    n_serves(E, DIR, STOPS),
    STEP = STOPS + DIST.
```


Implementation – #program base.
Elevator Instructions:
- fix the serving times: determine *where* and *when* the elevator has to serve
- determine when elevators have to switch direction
- determine the current direction for each time step
% For bidirectional elevators we need to know
% when they switch direction
switch_point(E, STEP) :-
bidirectional(E),
initial_direction(E, DIR),
furthest_along(E, DIR, DIST),
n_serves(E, DIR, STOPS),
STEP = STOPS + DIST.

- general elevator instructions → elevators act according to their predefined plan
- make sure to serve requests at the starting positions if there are any
- determine serving “where’s” and “when’s” → elevators have fixed schedule, calculations based on travel distances and previously made stops and moves
- set the switching directions time step → when elevator has reached target that was the furthest above or below
- make sure the elevator knows in each time step n which direction he has to move next

Implementation – #program step(t).

- derive `do` and `holds` predicates based on `serve_at` plans
- elevator moves if it does not serve and there are remaining targets
- carry requests along if they are not served yet
- update elevator positions after every `move`

```
% serve if the plan says so
do(elevator(E), serve, t) :-
    serve_at(E, F, t),
    holds(at(elevator(E), F), t - 1).

% move if there is something left to do
do(elevator(E), move(DIR), t) :-
    current_direction(E, DIR, t),
    not serve_at(E, _, t).
```


- entire incremental part is derived from the previously determined plan
- elevators only serve if the plan says “serve at this time step”
- if there is something left to do (`current_direction()`), the elevator has to move (no idling)
- if a request is not served, then it is carried along (if there was no serving on the request floor)
- update the position after each time step

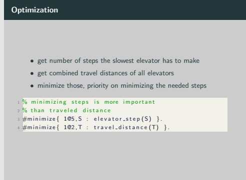
Optimization

Optimization

Optimization

- get number of steps the slowest elevator has to make
- get combined travel distances of all elevators
- minimize those, priority on minimizing the needed steps

```
% minimizing steps is more important
% than traveled distance
#minimize{ 1@5,S : elevator_step(S) }.
#minimize{ 1@2,T : travel_distance(T) }.
```

- get the maximum number of steps the slowest elevator has to make
- slowest elevator = elevator with most requests/longest distance
- sum up travel distances of all elevators
- first minimize the steps, then the overall travel distance

Results

Results

Results

- no timeouts
- slowest instance: test case 58 with 11927 ms
- overall time: 149028 ms

Results

- no timeouts
- slowest instance: test case 58 with 11927 ms
- overall time: 149028 ms

- ...