Rush Hour
An A* implementation

Fabian Pirklbauer, Sascha Zarhuber

Preconditions

Implementing an A* search algorithm

- Initialized using root node
  - put on open list

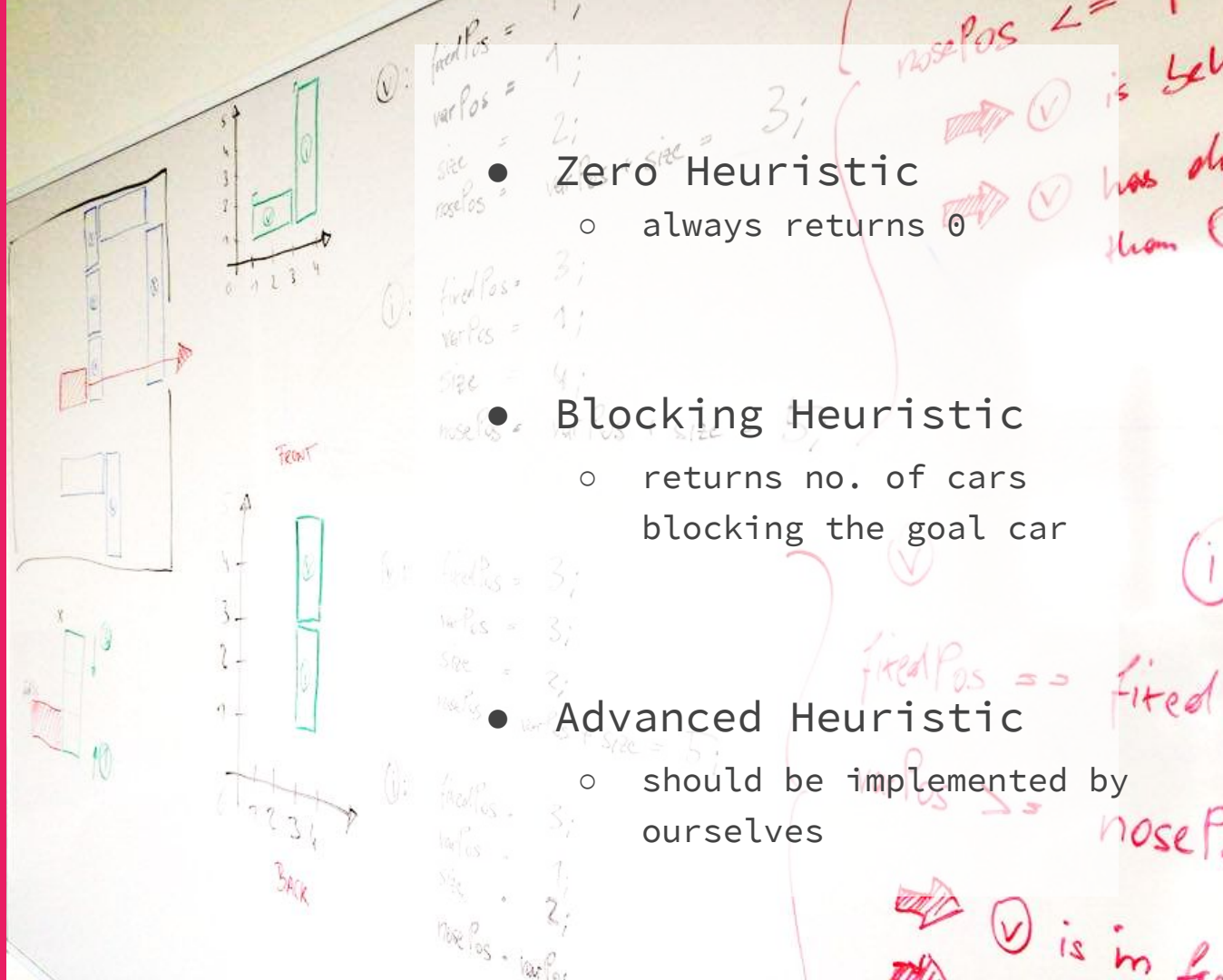
- For every successor:
  - Calculate heuristics
  - Add successor to open list, sort it based on heuristic value
  - Add parent node to closed list

Implementing an A* search algorithm

- Check if better alternative to current successor is already available on open list:
  - Else if not already on closed list, add it to open list

```
if (open.contains(successor)) {
      keepBetterNodeOnOpenList(successor);
} else if (!closed.contains(successor)) {
      open.add(successor);
}
```

Fun with heuristics



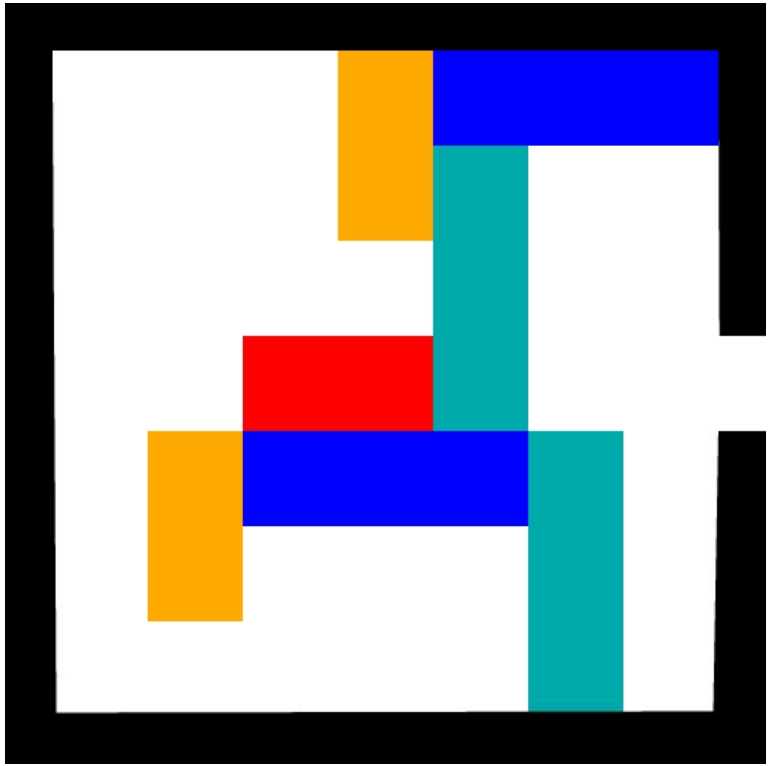
- Zero Heuristic
  - always returns 0
- Blocking Heuristic
  - returns no. of cars blocking the goal car
- Advanced Heuristic
  - should be implemented by ourselves

Advanced Heuristic approach

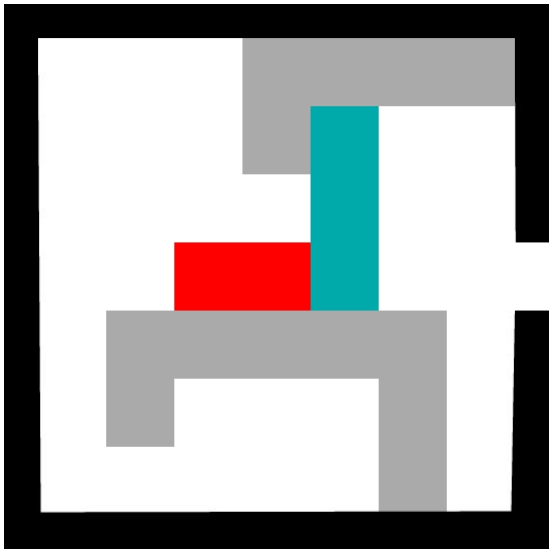
- BlockingHeuristic doesn’t consider:
  - Space between blocking cars
  - Orientation

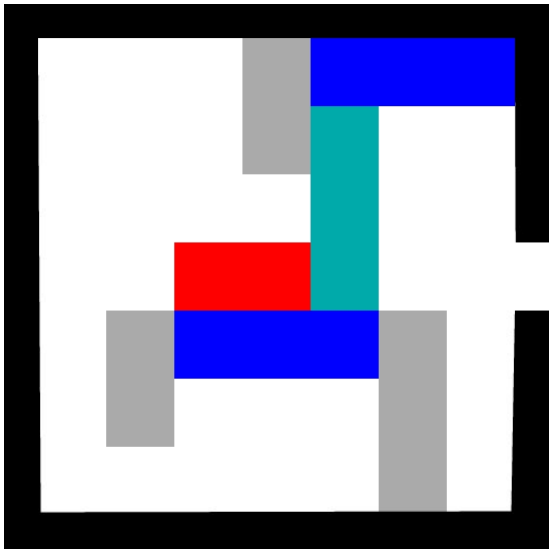
- We wanted to compensate this fact
  - Just because moving cars out of collision line doesn’t mean it’s a better/faster solution
  - Recursive approach to make use of data based on state

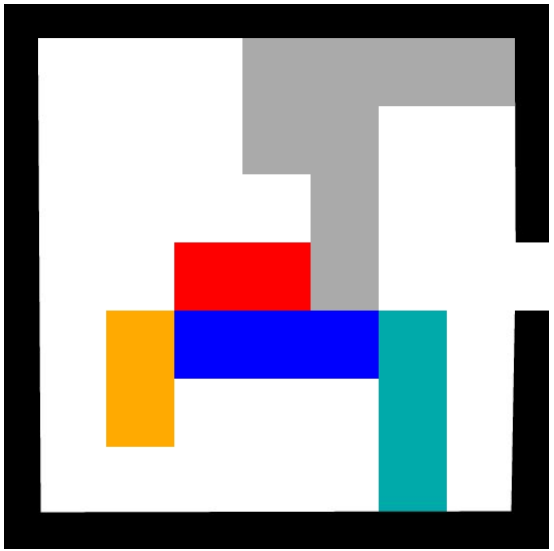
Sample



Sample







moveDown: 3

moveUp: 1

canMove? false

canMove? false

Move on to next

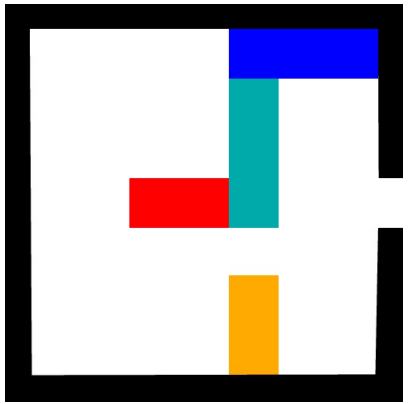
Brainteasers

- How to handle wall collisions?
  - Since we're always investigating both directions, we have to decide which way we expect to be the best

- We're not allowed to over-estimate
  - Return temporary `Integer.MAX_VALUE` as heuristic value
  - Gets ignored later when to decide which way to take

Brainteasers

- What to do when two cars align?
  - Can we “throw” needed space onward?
    Blocking car can only move in same direction as current car



Initial Heuristic Call

```
getValue(State state) {
    return 0 if state.isGoal()

    visited.clear()
    visited.add(goalCar)

    value = 1;

    for (getInitialBlockingCars() as car) {
        value += getBlockingValue(car, getRequiredSpace(goalCar, car))
    }

    return value;
}
```

Recursive Blocking Cars Calculation

```
getBlockingValue(car, requiredSpace) {
    visited.add(car)

    value = 1

    for (allCars as next) {

        continue if next == car
        continue if visited.contains(next)
        continue unless car.intersectsWith(next)

        forwardCosts = 0, backwardCosts = 0


        // Continues on next slide...
    }
```

Recursive Blocking Cars Calculation

```
for (allCars as next) {
        // Continues from next slide...

        if (!canMoveForward(car, next, requiredSpace.forward) {
            forwardCosts = getBlockingValue(car, getRequiredSpace(car, next))
        } else if (isWallBlockingAhead(car, requiredSpace.forward)) {
            forwardCosts = INFINITY
        }

        if (!canMoveBackward(car, next, requiredSpace.backward) {
            backwardCosts = getBlockingValue(car, getRequiredSpace(car, next))
        } else if (isWallBlockingBehind(car, requiredSpace.backward)) {
            backwardCosts = INFINITY
        }

        value += min(forwardCosts, backwardCosts);

    }

    return value;
}
```

Results

Board	ZeroHeuristic			BlockingHeuristic			AdvancedHeuristic		
name	**nodes**	**dpth**	**br.fac**	**nodes**	**dpth**	**br.fac**	**nodes**	**dpth**	**br.fac**
Jam-1	11587	8	3.066	8678	8	2.950	4026	8	2.661
Jam-2	24178	8	3.380	6201	8	2.820	4634	8	2.712
Jam-3	7814	14	1.789	5007	14	1.728	3931	14	1.695
Jam-4	3491	9	2.326	1303	9	2.061	378	9	1.762
Jam-5	24040	9	2.928	8353	9	2.583	1999	9	2.173

Demo

Thanks!

Questions?