AI Homework Project Report I: The Rushhour Puzzle

Rainhard Findling

17th April, 2012

Contents

Chapter 1

Solution to problem 1 of "Part I: Written Exercises"

1.1 a. Consistency of h

A heuristic is consistent if $h_1 \leq c + h_2$ is true for every heuristic value h_1 and h_2 and path cost c in the graph. This check has been done for every edge in the graph in table 1.1. As h is consistent for every edge in the graph, it can be considered at consistent.

n_1	$h(n_1)$	n_2	$h(n_2)$	$c(n_1, n_2)$	$h(n_1) \leq h(n_2) + c$
S	9	U	10	2	TRUE
S	9	T4	9	2	TRUE
T4	9	B4	7	2	TRUE
B4	7	T3	7	2	TRUE
T3	7	B3	5	2	TRUE
B3	5	T2	5	2	TRUE
T2	5	B2	3	2	TRUE
B2	3	T1	3	2	TRUE
T1	3	B1	1	2	TRUE
B1	1	G	0	1	TRUE
U	10	V	1	9	TRUE
V	1	G	0	1	TRUE

Table 1.1: The heuristic h is consistent for every node in the graph.

1.2 b+c. Search tree for Greedy Best First and A*

Greedy Best First relies on the heuristic function h only, as shown in the search tree in figure 1.1[1]. A* search combines a weighting function g with a heuristic function h to an evaluation function f, as shown in the search tree in figure 1.2.

[1]The expansion order is shown by the #-numbering.

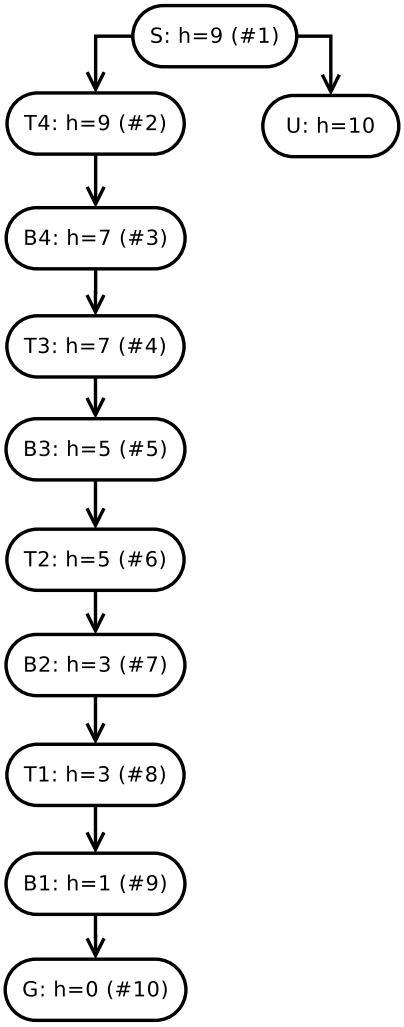


Figure 1.1: Search tree for applying Greedy Best First search to the graph.

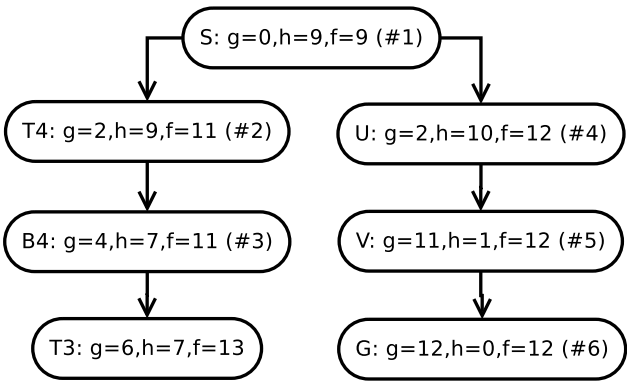


Figure 1.2: Search tree for applying A* search to the graph.

Chapter 2

Solution to "Part II: Programming"

2.1 Performance of the heuristics for the RushHour Puzzle

The heuristics for the RushHour Puzzle performed as expected:

- Using no heuristics (`ZeroHeuristic.java`) turns the A*-Algorithm into a Dijkstra-Algorithm. As the search is undirected then (yet informed), this results in expanding the most nodes in the graph. Calculating the heuristic values is computationally very cheap (it is constant).
- Using `BlockingHeuristic.java` results in expanding less nodes than for the Zero-Heuristic. Calculating the heuristic values is computationally more expensive than for the ZeroHeuristic.
- Using `AdvancedHeuristic.java`, the amount of expanded nodes decreases further, but calculating the heuristic values is again computationally more expensive than for the BlockingHeuristics.

Table 2.1 shows the output of `RushHour.java` when running the A*-Algorithm with all heuristics for all jams. The jams with index ≤ 0 are self implemented special cases which act as test functions for the later developed Advanced Heuristic. Some exemplary special cases are described in section 2.3.2.

2.2 Thoughts on search and the heuristics for the RushHour Puzzle

As clearly visible in table 2.1, for some solutions the effective branching factor b^* is very different than for some other solutions. For the solutions with a smaller b^*, either there have not been that much possibilities to chose amongst, or parts of the search tree have been pruned[1] during search due to the heuristic. For solutions with a bigger b^*, there were more possibilities to chose amongst, or parts of the search tree, that looked interesting, but turned out to be blind alleys, got expanded too.

What makes the RushHour Puzzle easy for humans? A human can detect very fast if a particular step in the RushHour Puzzle will take him towards solving the puzzle or not. Therefore humans have a quite good heuristic, which is most probably not even admissible. When the puzzle is almost empty, there are much possible steps to choose amongst. With

[1] Those nodes did not get expanded due to too high f-costs.

	ZeroHeuristic			BlockingHeuristic			AdvancedHeuristic		
name	**nodes**	**dpth**	**br.fac**	**nodes**	**dpth**	**br.fac**	**nodes**	**dpth**	**br.fac**
Jam--5	16065	6	4,833	5389	6	3,991	870	6	2,878
Jam--4	16065	6	4,833	5013	6	3,940	791	6	2,828
Jam--3	5219	5	5,314	951	5	3,701	256	5	2,773
Jam--2	5336	4	8,276	438	4	4,281	334	4	3,977
Jam--1	4439	3	16,087	58	3	3,463	58	3	3,463
Jam-0	44404	4	14,255	458	4	4,333	164	4	3,269
Jam-1	11587	8	3,066	8678	8	2,950	4026	8	2,661
Jam-2	24178	8	3,380	6201	8	2,820	1802	8	2,385
Jam-3	7814	14	1,789	5007	14	1,728	3931	14	1,695
Jam-4	3491	9	2,326	1303	9	2,061	298	9	1,709
Jam-5	24040	9	2,928	8353	9	2,583	1893	9	2,158
Jam-6	16046	9	2,792	7339	9	2,543	4554	9	2,402
Jam-7	56222	13	2,215	24529	13	2,068	21777	13	2,048
Jam-8	6470	12	1,957	5864	12	1,940	2560	12	1,797
Jam-9	5913	12	1,941	3704	12	1,860	2513	12	1,794
Jam-10	16197	17	1,677	12787	17	1,651	11621	17	1,641
Jam-11	6848	25	1,349	5918	25	1,340	4641	25	1,325
Jam-12	11686	17	1,642	7247	17	1,591	5309	17	1,559
Jam-13	72952	16	1,923	33949	16	1,827	20066	16	1,763
Jam-14	116381	17	1,901	44014	17	1,787	32904	17	1,755
Jam-15	3197	23	1,338	3171	23	1,337	3157	23	1,337
Jam-16	23072	21	1,534	17662	21	1,513	16097	21	1,506
Jam-17	19580	24	1,436	18287	24	1,432	13127	24	1,410
Jam-18	13881	25	1,392	12024	25	1,383	7171	25	1,352
Jam-19	3585	22	1,366	3533	22	1,365	3471	22	1,364
Jam-20	13881	10	2,464	4925	10	2,203	1832	10	1,975
Jam-21	1690	21	1,334	1590	21	1,329	1266	21	1,313
Jam-22	32371	26	1,423	22704	26	1,402	15221	26	1,378
Jam-23	19901	29	1,342	11978	29	1,316	9038	29	1,302
Jam-24	46313	25	1,468	43306	25	1,464	41699	25	1,461
Jam-25	82852	27	1,457	64644	27	1,443	55007	27	1,433
Jam-26	40855	28	1,397	34202	28	1,387	32080	28	1,384
Jam-27	21342	28	1,362	17927	28	1,352	16441	28	1,348
Jam-28	15591	30	1,316	11162	30	1,299	4682	30	1,257
Jam-29	38355	31	1,345	38111	31	1,345	37496	31	1,344
Jam-30	8610	32	1,264	7930	32	1,260	7022	32	1,255
Jam-31	32852	37	1,270	31236	37	1,268	29585	37	1,266
Jam-32	3292	37	1,184	2853	37	1,178	2680	37	1,176
Jam-33	37590	40	1,250	24507	40	1,235	16453	40	1,221
Jam-34	37544	43	1,229	36076	43	1,227	33867	43	1,225
Jam-35	34045	43	1,225	33326	43	1,225	32983	43	1,224
Jam-36	22760	44	1,207	18627	44	1,201	16381	44	1,197
Jam-37	15270	47	1,179	15231	47	1,179	13906	47	1,177
Jam-38	28560	48	1,192	24130	48	1,187	22552	48	1,186
Jam-39	24877	50	1,179	24361	50	1,179	24022	50	1,178
Jam-40	24467	51	1,174	22288	51	1,172	18299	51	1,167

Table 2.1: Output of `RushHour.java` for all tested RushHour Puzzle heuristics.

using a good heuristic, it's easy for humans to prune large parts of the search space and to only try a very small subspace of all possible actions – this will result in a low b^*. A computer program, which does not have a that good heuristic, will have to search through much larger parts of the search space. For the mentioned case of a almost empty puzzle with many possible actions, this leads to a very high b^*, compared to what humans will achieve. When the puzzle is almost filled, the bad heuristics of the computer program does not matter that much any more, as there are not that many actions to chose amongst any more. This will automatically lead to a smaller b^*. This can be seen in table 2.1: it is clearly visible, that with increasing depth of solution (which directly correlates to a more filled-up puzzle), the branching factor decreases for all heuristics.

Generally spoken: the bigger the average branching factor b of the graph and the worse the heuristic, the more effort has to be taken to solve the problem by search. For some particular problems, humans have a way better heuristic than most computer programs: in these cases, humans will be able to find solution with less effort than a computer program (it's rather "easy" for them). When there are few possibilities (lower b), even a program with a bad heuristics has to search through less possibilities, which makes search easier.

Searching with a computer can be improved by using better heuristics – specially non-admissible/non-consistent ones, as humans would do.

2.3 Constructing the Advanced Heuristic

2.3.1 Problem statement

Designing a heuristic for the RushHour Puzzle is easy – as long as it does not have to be admissible or consistent. As the Advanced Heuristic must be consistent, it cannot use the distance the red car still has to take to the goal[2] and similar metrics.

The Advanced Heuristic has been defined as follows: it should not only count the amount of cars C_1 that block the lane of the red car (as done by the BlockingHeuristic), but it should also count the amount of cars C_2 that hinder the cars in C_1 from clearing the lane. It should not matter how the cars in C_2 would have to move to really free the way, only the amount of cars in C_2 that have to be moved to free they way shall be counted.

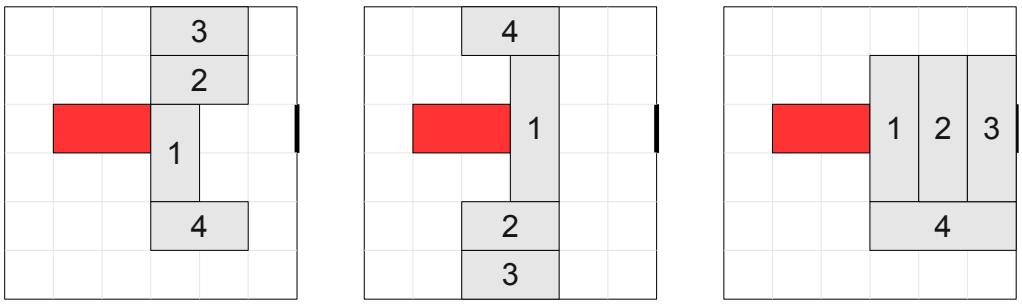
2.3.2 Special Cases

The following special cases might be a problem for a poorly designed heuristic. Figure 2.1a shows that car 1 is blocked by car 2 and 3 above, and by car 4 below. The best exit strategy is to move car 4 and car 1 without moving car 2 or 3. So a simple heuristic that counts how much cars C_2 in total block the cars C_1 is not enough: it might be enough to move only a part of the cars in C_2 to free the way. Figure 2.1b shows the same scenario, but car 1 has too less space above to move there. Figure 2.1c shows the same scenario again – this time the space above is empty, but the car still has too less space to move there. The heuristic must in both cases only count the valid exit, which in both cases is to move car 2 and 3.

To overcome the problem shown in figure 2.1a, a possible approach would be to always use the exit strategy, for which the least cars have to be moved. Figure 2.2a shows a special case for which this leads to a wrong amount of cars to be moved[3]. As exit strategy with

[2] Can be overcome by a single move if the lane is free.

[3] It is obvious that car 4-6 cannot be moved out of the way easily – this constellation just acts as an example for the problem class.

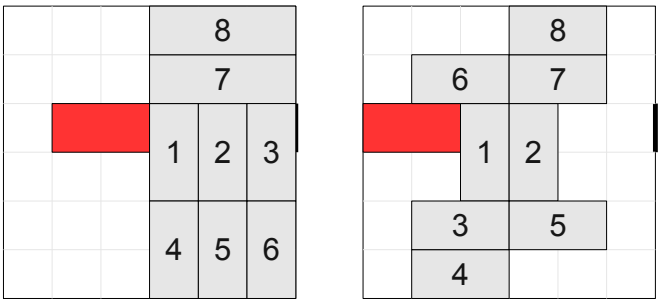


(a) **(b)** **(c)**

Figure 2.1: The heuristic must only count valid exit ways with least cars to move.

least cars to move car 1 has car 4 only. The same is true for car 2 and car 5, and for car 3 and car 6. The heuristic would propose to move car 4, 5 and 6 to free the way. Moving car 7 and 8 would indeed free the way with moving one car less in total. Therefore the usage of the exit strategy with the least cars to move is not a good idea.

Another approach would be to either take the above- or below-exit-strategy. This would solve the problem shown in figure 2.2. Figure 2.2b shows another special case, for which this approach will again lead to a wrong amount of cars to be moved. Car 1 and car 2 have to move into different directions for clearing the way will least moves possible. Therefore car 6 has to be moved out of the way of car 1, and car 5 has to be moved out of the way of car 2.



(a) **(b)**

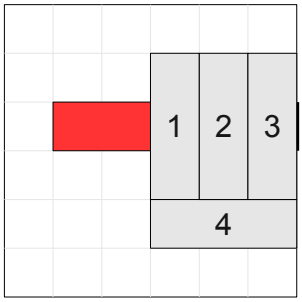
Figure 2.2: The heuristic must not simply take the least amount of cars to move (a), and it cannot take either the above or the below exit strategy only (b).

2.3.3 Implementation

The implementation of the Advanced Heuristic uses the following approach:

1. Let the cars C_1 blocking the lane of the red car vote for their exit strategies in an exit strategy voting list. Every car can therefore vote at maximum 2 times: one time for the exit above, and one time for the exit below. E. g. for figure 2.2b car 1 votes for $(3, 4)$ and for (7). Car 2 votes for (5) and for $(6, 7)$. For each voting, the amount m of cars that are part of the exit strategy and the amount n of cars that have voted for this exit strategy are important.
2. The exit strategy with the highest $\frac{n}{m}$ ratio gets applied, therefore it gets removed from the exit strategy voting list first. If more exit strategies have the same ratio, use the one with more cars involved first. E. g. for figure 2.2b this would either be (7) or (5). Every car that had previously voted for this exit strategy now has a free way to clear the lane. Therefore it can be removed from all other exit strategies it has also voted before. If there is an exit strategy no one votes for any longer, remove it from the exit strategy voting list too.
3. Go back to 2. until the exit strategy voting list is empty.

Figure 2.3, 2.4 and 2.5 demonstrate how the developed Advanced Heuristic chooses exit strategies. The total amount of cars part of those exit strategies that get applied in the end can be added to the value the BlockingHeuristic would obtain. The resulting value is what the AdvancedHeuristic returns as heuristic value for a given puzzle.



(a)

Exit	Votes	Ratio	Nr.
(4)	1,2,3	3	1

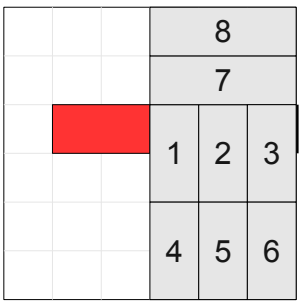
(b)

Figure 2.3: Choosing exit strategies for figure 2.1c using the Advanced Heuristic

2.3.4 Admissibility/Consistency

Why are the heuristics admissible/consistent?

BlockingHeuristic: when a car blocks the lane, it first has to be moved out of the lane for sure. Therefore it is admissible as it never overestimates the costs to the goal. The heuristic is also consistent, as when the specific blocking cars has been moved out of the way it always returns a heuristic value that is reduced by 1 compared to when the car blocked the lane.

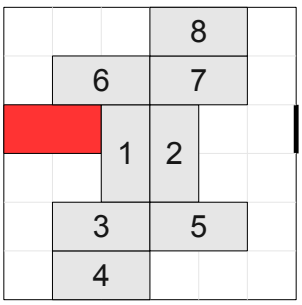


(a)

Exit	Votes	Ratio	Nr.
$(7,8)$	1,2,3	$\frac{3}{2}$	1
(4)	1	1	-
(5)	2	1	-
(6)	3	1	-

(b)

Figure 2.4: Choosing exit strategies for figure 2.2a using the Advanced Heuristic



(a)

Exit	Votes	Ratio	Nr.
(7)	1	1	1
$(3,4)$	1	$\frac{1}{2}$	-
(5)	2	1	2
$(6,7)$	2	$\frac{1}{2}$	-

(b)

Figure 2.5: Choosing exit strategies for figure 2.2b using the Advanced Heuristic

AdvancedHeuristic: when cars C_1 are blocking the lane of the read car, and their exit way is blocked by other cars C_2, the least possible moves of C_2 to free the ways for all cars in C_1 to clear the lane are admissible, as there cannot be a solution with less moves to take in total – and it never overestimates the costs therefore. The tricky thing hereby is to find out which the least moves of C_2 are. The heuristic is also consistent, as when the exit way with the least moves for a car in C_1 gets blocked by one car less, it will always return a heuristic value reduced by 1 compared to when the way was still blocked by 1 more car.

2.3.5 Results

The AdvancesHeuristic proofed to achieve better results (measured by the amount of expanded nodes) than the BlockingHeuristic, and it also proofed to not lead to false results (due to not being admissible/consistent), as seen in table 2.1. The step from BlockingHeuristic to AdvancedHeuristic did not improve the results very much, compared to the step from ZeroHeuristic to BlockingHeuristic. The only exceptions from this are the special cases (jams with index ≤ 0 in table 2.1). Those are special cases like the ones discussed in section 2.3.2, for which the BlockingHeuristic cannot provide very good heuristic estimates.

Generally spoken, the increased effort used in an more advanced heuristic must be allegeable by it's outcomes (otherwise an A*-Search could also be used as heuristic as it would always return the exact value). If the computational effort of the heuristic is too high, the target of the heuristic has missed it's target, which is a fast estimate, not an exact value.