Heuristics Bypassing with Non-Consistent Heuristics in A*

Elon Dagan

Faculty of Data & Decision Sciences

Technion

Israel

Abstract

Given multiple possible admissible heuristics, one of the most common ways to use them for optimal search is to take their maximum. Although this approach guarantees minimum possible number of nodes expansions (with respect to the available heuristics in hand), this approach may result in overall more time to find the optimal solution due to the computation time it requires to calculate the heuristic value of each heuristic for each state. In this paper, I suggest different method to use multiple heuristics that might result in more node expansions but with better time performance overall.

1 Introduction

For the goal of optimal search in a state space, A* with admissible heuristic guarantee that the expanded nodes in the search process are both sufficient and necessary to find the optimal solution [Dechter and Pearl, 1985]. In the search process, A* is expanding all the nodes with an f value that is smaller or equal to the cost of the optimal solution [Hart, et al, 1985]. Thus, the higher values an admissible heuristic is returning, the lower the number of nodes that will be expanded and therefore the overall search time. From that it follows that given several admissible heuristics, using their maximum as a heuristic will guarantee the minimum number of nodes expansion possible (that is possible to achieve with the given heuristics). This would be the best possible approach if the computing time of all the heuristics was negligible, and since in general, non-trivial heuristics computing time is not negligible, A* with max heuristics is often not the best approach.

There are two main drawbacks of using the maximum of several heuristics that can makes it not the best approach. The first is that for some nodes, neither of the heuristic can achieve an f value large enough to surpass the cost of the optimal solution, and for these nodes, computing the heuristics didn't contribute to finding the goal faster since they will be expand in the future. The second is that for some nodes, we could obtain an f value that is bigger than the real cost of the solution by computing only one of the heuristics, but since we don't know what the real cost of the solution is until we solve the task, we need to also compute the rest of the heuristics. These drawbacks can rise in two different cases. First is when we have several heuristics that are in general equivalent both in computing time and returning estimated cost value (i.e. in each state any of the heuristics can be the one obtaining the maximum). It can also rise in the case of several heuristics where each of the heuristic is dominant and more expansive then the other (i.e. $h_1 \leq h_2 \leq h_3$, $t_1 \leq t_2 \leq t_3$). These drawbacks can also rise somewhere between both cases.

There has been research trying to overcome these drawbacks such as LA* and RLA* algorithms [Karpas *et al*, 2013]. The LA* algorithm does not compute all the heuristics together, instead each time a node appears from the OPEN list, we compute only one heuristic – the heuristic that has not be computed yet and it's the next in line. This method guarantees same results as using the maximum of the heuristics and not worse (and most of the time better) time performance. This algorithm handles the second A* drawback with the second case – if for some node, h_1 is enough to surpass the real cost, then h_2 will never be use, but it's not handling the first drawback – for nodes that both $h_1$and h_2 will not surpass the f value, it is wasting time computing both heuristics. The RLA* on the other hand tackles both issues for the second case (dominating heuristics) – like LA*, it computes only one heuristic every time we pop a node, but on top of that, it also tries to predict when a node re-emerges from the OPEN list, if $h_2$could be helpful, and when it decides it won't, it don't waste time computing it. Although RLA* tackles both of A* drawbacks, it has some limitations. One of them is that it could achieve better performance then LA* only for domains and heuristics that hold that $p_n b(n) < 1$, when p is the probability of h_2 being helpful, based on the number of times it has been helpful so fur.

In this paper, I will introduce a different version to LA* and RLA*, called Predictive Lazy A* (PLA* hereafter). PLA* like RLA*, tackles both of A* drawbacks for the case we have dominating heuristics, but is suitable in cases that RLA* isn't.

2 Predictive Lazy A*

2.1 Notations

Throughout this paper, I denote the optimal solution cost of a given task as c^* and the average computation time of the i'th heuristic as t_i. For simplicity, I assume we have two heuristics h_1 and h_2 such that $t_1(n) < t_2(n)$ and for most nodes, $h_1(n) \leq h_2(n)$. In a similar (but slightly different) way to the LA* paper, I define two types of nodes – *good* nodes and *bad* nodes. a node n will be call *good* if $f_1(n) \leq c^*$ and $f_2(n) > c^*$. And node n will be call *bad* if $f_1(n) < c^*$ and $f_2(n) < c^*$. I also denote the probability that node n is good with $p(n)$ (or p in general).

2.2 The Algorithm

PLA* performs exactly the same as RLA*. When a node is expanded, for each of its children we compute only f_1 and push him into the OPEN list. When a node is popped from OPEN, we try to predict if n is a *good* or *bad* node. That is, since n emerged from OPEN, we know (assuming n is not the goal state) that $f_1(n) < c^*$ and we want to predict if f_2 will be larger or smaller then c^*. The difference between the algorithms is the way we make the prediction. In PLA*, every node has its own values to the features of a prediction model, and every node gets a prediction made specifically for him. This prediction is made by a pretrained logistic regression model and a learned threshold th. Given the features values of a node n, using the model we obtain a probability estimation $p(n)$. Then, if $p(n) > th$, PLA* will compute f_2 and will push the node back to OPEN and if $p(n) \leq th$ then it will expand the node without computing f_2.

2.3 Evaluating P

To evaluate p for each node, PLA* use a pretrained logistic regression model and a learned threshold. The training of the model is done prior to solving the tasks and is performed as follows. First, we solve k problems with A* using only h_1. After solving each task, we iterate the resulted CLOSED list and for each node we create a training sample with 5 features - 1 (bias), $h_2(initial\ state)$, $n.path\_cost$, h_1(n), $h_1(n.parent) - h_1(n)$ (will be called *n.bias, n.pc, n.h1, n.h2, n,diff* hereafter). the label of n is 1 if $f_2(n) \geq c^*$ and 0 otherwise. All of these features (except of the second, which computed only once in the beginning of a search), are available during a search without performing more computation then A* is already perform by keeping each feature of a node as its attribute (the same way each node hold the path cost used to evaluate f).

The reason for iterating over the CLOSED list is that each node in that list has been expanded during the search process of A* (that used h_1), thus we know that for each node n in this list, $f_1(n) < c^*$. Therefore each of the nodes in EXPLORED is either *good* or *bad*. The nodes that are left in the OPEN list when the search process ended are not important for the training, since for them f_1 was enough to surpass c^*, and for these nodes, PLA* will not have to decide whether to compute f_2 or not since they won't reappear after being pushed to OPEN.

The purpose of the second feature of the model is to try capturing the difference in the importance of the other features based on c^*. For example, for two different tasks, one with $c^* = 10$ and the other with $c^* = 25$, there is a big difference how valuable the node 'path cost' or the 'h_1' features are. Since we don't know c^* in advance, the model uses the best prediction to it, $h_2(initial\ state)$. Although the ideal is to create a specific model for each h_2 evaluation of the initial state, which is likely to results in better performance, in this paper I am content with just add the initial evaluation of h_2 to the features so I could use one model for all possible tasks, but further improvements to PLA* using multiple models is possible.

2.4 Choosing a Threshold

Choosing the best threshold to use with the model on a given domain is based on the computation time of both heuristics (t_1 and t_2), the ratio between *good* nodes to both good and bad nodes ($\frac{(good\ nodes)}{good\ nodes+bad\ nodes}$) and the model FN (false negative) and TN (true negative) values. The goal is to find a threshold that will perform batter then LA*. Consider some node n, notice that if the model predicts that n is good, whether n is good or bad, both models spend the same amount of time (PLA* predicted good so h_2 was computed and LA* always compute h_2). The difference between the two algorithm occurs when PLA* predicts that n is bad when n is good, and when PLA* predicts that n is bad when it's indeed bad. In the first case, PLA* save t_2 time and waste at least $b(n)t_1$. In the second case, PLA* saves t_2 time. Note that although $b(n)t_1$ is a lower limit to the time PLA* pay, since we don't know what the result of the children of the expanded node will be (f_1 might be enough for them to not emerged from OPEN), we are satisfying with this value, since we only trying to find a threshold that maximize the equation below. The gain of PLA* on a good or bad node result in the following equation:

$$p(n\ is\ good) \cdot p(pred\ good \mid n\ is\ good) \cdot 0\ +$$
$$p(n\ is\ good \cdot p(pred\ bad \mid n\ is\ good) \cdot (t_2 - b \cdot t_1)\ +$$
$$p(n\ is\ bad) \cdot p(pred\ good \mid n\ is\ bad) \cdot 0\ +$$
$$p(n\ is\ bad) \cdot p(pred\ bad \mid n\ is\ bad)$$

Which is equal to

$$p(n\ is\ good) \cdot p(pred\ bad \mid n\ is\ good) \cdot (t_2 - b \cdot t_1)\ +$$
$$p(n\ is\ bad) \cdot p(pred\ bad \mid n\ is\ bad)$$

And since we have an estimation to $p(n\ is\ good)$, and since $p(pred\ bad \mid n\ is\ good)$ and $p(pred\ bad \mid n\ is\ bad)$ are captured by FN and TN, respectively, we obtain the equation:

$$p(n\ is\ good) \cdot FN + \big(1 - p(n\ is\ good)\big) \cdot TN$$

Therefore, the threshold we would like to use is the threshold that obtain the FN and TN values the maximize this equation. This can be done in the training phase by using different thresholds, and test the performance of each of them on the model.

	computation time			Expanded nodes
Domain	h_c	h_{lmc}	h_{lmc}/h_c	h_c/h_{lmc}
puzzle	7.653e-04	0.004	5.6	3.46
Blocks	4.414e-06	0.003	875.4	3.25
Trucks	8.431e-05	0.004	53.6	3.31
Vacuum	6.466e-05	0.005	83.5	8.6
Elevator	4.364e-06	0.008	1993.2	4.37

Table 1: comparison between the LMC and the domain dependent heuristic for each domain.
the right columns show the ratio between the number of expanded nodes on average of the heuristics

	h_1			h_2			LA^*				PLA^*			
c^*	time	explored	Frontier	time	explored	Frontier	time	explored	Frontier	Used h_2	time	explored	Frontier	Used h_2
20	1.5	1124	649	2.43	350	259	2.76	350	259	593	1.65	1124	649	0
24	7.3	4038	2232	9.03	1227	785	10.39	1226	785	1979	7.64	1875	1147	1021
16	0.3	309	200	0.54	81	65	0.63	81	65	138	0.4	309	200	0
20	2	1410	853	2.94	442	288	3.38	442	288	722	2.14	1410	853	0
22	2.4	1690	994	3.43	446	300	3.79	446	300	739	2.69	1590	958	83
22	3.1	2135	1266	3.94	510	355	4.29	510	355	853	3.43	2007	1210	97
24	11	5384	2906	12.37	1711	1106	13.97	1710	1105	2751	10.61	2560	1569	1447
12	0.1	67	46	0.13	21	21	0.14	21	21	36	0.09	67	46	0
26	25.3	8723	4458	18.74	2268	1511	20.49	2267	1510	3735	15.86	2413	1581	2481
16	0.2	162	108	0.33	47	39	0.36	47	39	79	0.22	162	108	0
22	5.9	3012	1729	6.44	953	622	7.42	953	622	1559	6.61	2191	1302	537
20	1.6	1204	739	2.38	343	236	2.72	343	236	571	1.81	1204	739	0
18	0.5	409	261	0.8	109	82	0.88	109	82	184	0.54	409	261	0
24	13.6	5490	2940	14.3	1772	1111	15.65	1771	1111	2809	11	2054	1271	1549
12	0.1	57	43	0.09	15	13	0.1	15	13	23	0.09	57	43	0
16	0.4	311	208	0.63	91	66	0.74	91	66	151	0.45	311	208	0
22	1.9	1325	742	2.68	305	215	3.02	305	215	507	2.13	1289	720	29
22	4	2359	1365	6.11	774	508	6.77	774	508	1236	4.52	2037	1208	223
18	0.3	252	159	0.36	44	34	0.42	44	34	74	0.37	252	159	0
20	1.9	1239	770	2.94	386	263	3.28	386	263	635	2.1	1239	770	0
20	2.3	1484	874	3.76	544	353	4.3	544	353	869	2.48	1484	874	0
26	23.3	8080	4214	18.53	2011	1333	20.47	2010	1332	3310	16.65	2153	1407	2270
18	0.7	478	288	1.12	131	91	1.24	131	91	215	0.77	478	288	0
20	0.7	510	300	1.16	127	88	1.18	127	88	206	0.75	510	300	0
18	0.6	433	273	0.84	103	82	0.91	103	82	178	0.66	433	273	0
20	2.8	1686	1038	4.37	592	386	4.92	592	386	950	3.05	1686	1038	0
Tot	**113.9**	**53371**	**29655**	**120.4**	**15403**	**10212**	**134.2**	**15398**	**10209**	**25102**	**98.7**	**31304**	**19182**	**9737**

Table 2: 8-puzzle results. The 'used h_2' column shows the number of nodes for which the algorithm used both h_1 and h_2

2.5 Generalization

PLA* can be generalized to multiply heuristic in two different cases and in both the generalization is rather straightforward. The first case is that we have several heuristics the can 'act' as h_2. In this case in the training phase, instead of checking only if $f_2 > c^*$, we can perform that check for each of the heuristic in hand, and use logistic regression with multiple labels. During search, we can choose out of the heuristics that obtained probability that is bigger than the threshold, the one with the biggest probability. The second case is when we have several heuristics that each dominant and get dominant by the other (e.g. for 3 heuristic - $h_1(n) \leq h_2(n) \leq h_3(n)$). In that case, we can train a model the same way as we train for two heuristics, but now each model is train on a different pair (e.g. for 3 heuristics, we train a model on h_1 and h_2, and another model on h_2 and h_3). Then, we can choose which model to use based on the latest heuristic that was used (e.g. if we already computed h_2 and h_1, then will use the model that predicts with h_2 if h_3 will be helpful)

3 Empirical Evaluation

In the empirical evaluation, I test the performance of PLA* against LA* using h_1 and h_2, A* using only h_1 and A* using only h_2. Since the requirement for RLA* to perform better then LA* does not meet in the domains and heuristic I used, RLA* is not included in the results. For this part, for the more informative and more expensive heuristic h_2 I use the LM-cut heuristic (hereafter will refer as h_{lmc}), and for the less informative and less expensive heuristic h_1 I use a domain dependent heuristic for each domain (hereafter will refer as h_c). The purpose of using domain dependent heuristic as h_1 is just to serve as an example and any other domain independent heuristic can be used instead. Table 1 show a comparison between the two heuristics for each domain.

3.1 The Puzzle Domain

First, I will present the results for the 8-puzzle domain. In this domain, the threshold that has been used is 0.95. this high value is the result of two properties of this domain - h_1 is not much cheaper then h_2 (h_2 is approximately 5 times more expensive), and the branching factor of the domain is relatively low (approximately 2.7). this resulted in that for computing h_2 will be beneficial, PLA* need to be almost sure that n is a $good$ node. Table 2 depicts the results for 20 random tasks. In the table we can see how this threshold is reflected. For most of the tasks, PLA* results (in terms of explored and frontier nodes and number of heuristic computation) are exactly the same to A* using h_1. These tasks are the ones with solution length the is relatively small. On the other hand, for tasks with larger solution cost, since A* with h_1 expanding a lot more nodes, PLA* is confident about a lot of nodes that h_2 can be helpful, and as a result, in these tasks, computing h_2 for large number of nodes. We can also see that PLA* managed to compute h_2 for a lot less nodes then LA* and still expanding just a little more nodes then LA*.

3.2 all domains

Next, I tested the performance of PLA* on the rest of the domains. Table 3 depicts the coefficients of the model used in each domain, and table 4 the accuracy, precision and recall of the model.

For each domain, the logistic regression model was trained on 25 random tasks. Although training on tasks with different solution lengths can lead to better generalization and therefore better performance, I used random tasks which their solution length is not known in advance (and thus, it is likely that there are tasks in the training set with the same solution

		Features weight				
domain	$good/bad$	Bias	$n.h2$	$n.h1$	$n.pc$	$n.diff$
puzzle	0.78	-2.6e-05	-6.6e-02	4.4e-01	5.4e-01	1.6e-01
Blocks	0.91	3.5e-04	6.0e-03	9.6e-01	2.19	2.0e-01
Trucks	0.70	1.18	0.17	-0.65	0.09	-0.53
Vacuum	0.97	-1.35	-0.49	0.74	0.68	-0.09
Elevator	0.85	1.6e-04	-2.0e-01	-7.8e-02	5.6e-01	-2.1e-01

Table 3: For threshold of 0.5 (default)

	Model test results		
domain	Accuracy	Precision	recall
puzzle	0.84	0.86	0.95
Blocks	0.93	0.95	0.97
Trucks	0.77	0.79	0.92
Vacuum	0.97	0.97	0.99
Elevator	0.89	0.91	0.98

Table 4: model performance

	Problems solved				Average Planning Time (seconds)			
Domain	h_{LMC}	h_{custom}	LA^*	PLA^*	h_{LMC}	h_{custom}	LA^*	PLA^*
puzzle	45	45	44	**54**	6.667	6.667	6.818	5.555
Blocks	32	44	68	**69**	9.375	6.818	4.411	4.347
Trucks	16	23	23	**33**	18.750	13.043	13.043	9.090
Vacuum	71	50	71	**79**	4.225	6.000	4.225	3.797
Elevator	12	12	17	**18**	25.000	25.000	17.647	16.667
Overall	176	174	223	**253**	12.803	11.505	9.2288	7.891

Table 5: Problem solved in each domain in 5 minutes

cost while tasks with other solution cost that don't appear). The reason for using random tasks is to shorten the training phase which will become much longer if searching for different tasks with different solution lengths was included.

Furthermore, all models perform significantly better than a random model which predicts p based on the good nodes to both good and bad nodes ratio. For example, in the 8-puzzle domain, 74% of the nodes in CLOSED are $good$ nodes, and a random model that predict 1 (h_2 can be helpful) with probability 0.74, the accuracy should be 0.61 (probability of popping a good node square plus probability of popping bad node square, i.e. $0.74^2 + 0.26^2$), while the logistic regression obtain accuracy of 0.84

In the empirical evaluation, each algorithm was given the same 100 tasks to solve in the same order. Table 5 depicts the total number of tasks solved by each algorithm in each domain in 5 minutes, as well the average computation time it spends on each task

As we can see, PLA* perform better in all domains (in some more than others). These results become more significant as we are required to solve more problems, such that for enough problems, even adding the training time to the solving time, will results in better performance than the other algorithms

4 Conclusions

As seen in the empirical results, PLA* is able to achieve good results, and that despite the fact that are ways to improve even further the results, such as specific model for each possible h_2 evaluation of the initial state, train on a bigger train set and pick the tasks that will be included in the train set rater then choose them randomly. All this method was not included in

the model used in this paper for simplicity, but definitely can be used, when suitable, based on the requirements of the specific problem in hand. More than that, the logistic regression model is not the only model that can be used. The only requirement for the model is to keep the model simple, so the prediction phase will be cheap, since in is performed for every node that is emerged from the OPEN list.

Although the good results, PLA* has some limitations. The first and most obvious one is that the algorithm require a training phase before it is ready to solve tasks. This training phase can be expensive for some domains, especially when each task could take a long time to solve. Another limitation of the algorithm is that it achieves the best results when LA* is performing well. When LA* is perform worse then one of the heuristics, it is most likely that also PLA* will predict worse.

In conclusion, PLA* offer another and possibly a better way to use multiply heuristics that are dominating each other, and despite his limitations, can be the best choice in certain domains with appropriate heuristics.

5 Acknowledgments

The implementation of the landmark heuristic (H_{lm}) and the landmark cut heuristic (H_{lm_cut}) was taken from the '$pyperplan$' repository on GitHub

References

Dechter, Rina, and Judea Pearl. "Generalized best-first search strategies and the optimality of A." *Journal of the ACM (JACM)* 32.3 (1985): 505-536.

Hart, Peter E., Nils J. Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths." *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968): 100-107.

Tolpin, D., Beja, T., Shimony, S. E., Felner, A., & Karpas, E. (2013). Towards rational deployment of multiple heuristics in A*. In *Proceedings of the International Symposium on Combinatorial Search* (Vol. 4, No. 1, pp. 2