Creating Heuristics by Machine Learning

Robert Holte

Computing Science Department

University of Alberta

2011, 2013 Workshop Topics

- Using machine learning to configure a parametric planning system (including portfolio).
- Learning control knowledge for a planner (e.g. bad causal links).
- Learning action models, cost models.
- Learning plan rewrite rules to improve plan quality (e.g. shortcut learning).
- Learning heuristic functions.

Learning and Planning

GPS – The General Problem Solver

- Original domain-independent planner (1959)
- “Means-Ends Analysis”
  - Find the most important difference between current state and the current goal.
  - Choose an operator that can reduce this difference.
  - If the operator can be applied to the current state, do so.
  - If not, recursively find a plan to achieve the preconditions of the operator (these become the current goal).
- Declarative control structure called a “table of connections” (which operators could be used to reduce which differences; often also specifies a difficulty order of differences).
- 1960 paper looked at learning the control knowledge. Work on this continued through the 1970s but never succeeded.

STRIPS (1971)

- More than just a language. Continued the GPS tradition of domain-independent planning (1971).
- Used GPS's means-ends analysis as its planning method.
- 1972: MACRO-OP extraction from experience (learning).

Aside: Nilsson (1971) had the idea of action landmarks (he called them "key operators").

Plan Re-use/Adaptation

- STRIPS macro-ops were very much plan re-use, as opposed to what we now understand as macro-operators.
- Plan re-use became popular in the 1980s starting with the "problem-solving by analogy" of Jaime Carbonell (1981).

Learning Control Knowledge for Production Systems

Production Systems

- A state is a vector of values (usually fixed length).
- An operator is a production rule (LHS → RHS).
- Forward-chaining from a start state until a goal condition is satisfied.
- More a programming language than a domain-independent planning system, but it was non-deterministic so strongly resembled a search system.
- Introduced in A.I. around 1967
  (invented by Emil Post in 1943 as a universal model of computation).

Waterman (1970)

- Ordered set of production rules.
- Starting from a set of productions that played draw poker randomly, it learned when it was appropriate to take each action (fold, call, or bet a certain amount).
- Learning involved modifying existing production rules or creating new ones (and inserting them at the correct place in the order).

Kling (1971)

- First application of reasoning by analogy to problem-solving (theorem-proving).
- Idea: to prove a new theorem, reuse the lemmas, rules of inferences etc. that were needed to prove a similar theorem.
- i.e. use experience to reduce the branching factor (or at least to order the branches, like preferred

Late 1970s

- Brazdil, Langley, Mitchell, Stolfo – learning the conditions under which to apply a given rule (or, when NOT to apply it – Kibler & Morris 1981)
- Vere 1977 (and others) – learning production rules from “before/after” examples.

Aside: 1977 IJCAI paper on mutexes and regression planning (Dawson & Siklossy).

Learning Evaluation Functions for State-Space Search

Arthur Samuel (1956)

image from: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/ibm700series/impacts/

Learning an Evaluation Function

- Game-playing is not planning/search, but…
- Forward-chaining state-space search guided by an evaluation function.
- Evaluation functions in games are similar to what is needed for greedy best-first search as used in today's planners (as opposed to the heuristic functions for A*/IDA*).
- Some key ideas have been transferred (e.g. iterative deepening, transposition tables).
- However: Long training phase, single domain (GGP is the exception).

Samuel (1959)

- Evaluation function was a polynomial, used ML to
  - Select a subset of the features to include in the polynomial
  - Set the coefficients
- Learning was based on the difference between the value of a node and the value of the best node seen below the node (like TD learning).
- Therefore training data was readily available during the self-play of the training phase.
- Updates were made online.

Buro (1995)

- Buro’s ProbCut took this one step further.
- Based on the idea that the value k steps below a node would be highly correlated with the node’s value.
- Learned a function predicting the value k steps below a node (for some fixed k), and used that, instead of lookahead, to prune the node.
- Training was offline, based on actual lookahead.

State-space Heuristic Search

- Graph Traverser (now called greedy best first search) – Doran and Michie, 1966. “Satisficing search”. Anticipated learning the heuristic but no implementation.
- A* – 1968 (Ira Pohl’s thesis 1969).
- Subsequent work looked at generating heuristics by abstraction, not learning.
  - Somalvico & colleagues (1976-79)
  - Gaschnig (1977-79)
  - Pearl (1979-1984)
- First work on learning an evaluation function was by Larry Rendell (1977-1983). Instead of the usual heuristic “cost to go” function, his evaluation function estimated the probability that a state was on a solution path.

Creating Heuristics by Machine Learning

Learning a Heuristic = Regression

- We want machine learning to create a function, h(s), that takes a state (s) as input and produces a number (estimate of distance from s to goal) as its output.

- Training data will be a set of pairs (sk,yk), where yk is the cost of a path from sk to goal (ideally an optimal path).

How Do We Get the Labels (yk) ?

- There are a variety of ways of generating the states (sk) for training, but how do we get the labels (costs) for them (yk)?
- Two approaches:
  - Use search to solve sk
  - Use a method to predict the solution cost for sk

Two Scenarios

1. Long training time in order to create a heuristic that will solve subsequent problems very quickly.

2. Solve a single instance as quickly as possible.

Fundamental Paradox?

1. If the initial system is already good enough to solve arbitrary problems in a reasonable amount of time, there is not much to be gained by learning.
2. If the initial system is not that good to begin with, how is it possible for the training data it generates (solutions to “easy” problems) to enable it to learn to solve “hard” problems?

Part 1 – One-Shot Systems

- If the initial system is already good enough to solve arbitrary problems in a reasonable amount of time, there is not much to be gained by learning.
- "One-shot" systems create a training set by solving problems with the given heuristic, then apply learning to produce their final heuristic.
  - Ernandes and Gori (2004)
  - Samadi, Felner, and Schaeffer (2008)

Ernandes and Gori (2004)

15-puzzle

- Initial heuristic (h0): enhanced Manhattan Distance.
- Training: 25,000 random (solvable) instances solved using h0.
- Solving the training examples took 100 hours, learning the neural net took another 200 hours.
- Features for learning: binary state variables.
- Result:
  - 3.5% suboptimal
  - 500x speedup over h0

Samadi et al. (2008)

15-puzzle

- Initial heuristic (h0): best known PDBs.
- Training: 10,000 instances from random walks backwards from the goal of all lengths up to a hand-picked maximum. Solved using h0.
- Features for learning: MD, best known PDBs.
- Result:
  - 3.3% suboptimal
  - 8.3x speedup over h0

Fundamental Paradox(?), Part 2

If the initial system is not that good to begin with, how is it possible for the training data it generates (solutions to “easy” problems) to enable it to learn to solve “hard” problems?

Solution: Bootstrapping

- Larry Rendell's Ph.D. (1981) solved this "paradox" by a method I call bootstrapping.

- Repeat several times:
  - Use the current heuristic to solve whatever problems it can.
  - Those solved problems provide training data to a learning algorithm that produces a new heuristic.

Bootstrap – Input/Output

- Input:
  - initial (weak) heuristic h0,
  - a set of unsolved instances,
  - time limit,
  - features (we found small PDBs were excellent features, did not have to be carefully chosen)
- Output: h, a heuristic much better than h0

Bootstrap – More Details

- Heuristic search is given a time limit for solving each instance.
- When an instance is solved, all states along the path are added to the training set.
- Require at least M instances to be solved with the current heuristic before learning is invoked to create a new heuristic.
- If fewer than M are solved:
  - at the start: use random walks to generate very easy instances, and successively more difficult ones, until M of the original instances can be solved. (see next slides)
  - otherwise: double the time limit and continue with the current heuristic.

Harder Problems

- What to do if the initial heuristic is so weak that none of the given instances can be solved?

  IDEA: Generate instances using random walks of the "right" length.

  - Do BFS backwards from the goal up to a time limit. Find the length (L) of the average random walk that generates states not reached by BFS. Use multiples of L as the lengths of successive random walks.

Random Walk – Details

- Generate 500 instances using random walks of length L, and apply Bootstrap to them.
- As usual, Bootstrap may require multiple iterations and/or to increase the time limit to solve the random walk instances.
- If Bootstrap succeeds, test if the new heuristic can solve enough of the user-given instances.
  - If yes, apply Bootstrap to the user's instances.
  - If no, increase walk-length by L and repeat.
- If Bootstrap fails on the random walk instances, apply it to the user's instances.

24-Pancake, Random Walks

Length	Solved	Cost	Optimal	Subopt.	Nodes generated
5	500	4.98	4.98	0.0%	89

24-Pancake, Random Walks

Length	Solved	Cost	Optimal	Subopt.	Nodes generated
5	500	4.98	4.98	0.0%	89
10	392	9.21	9.19	0.2%	148,405

24-Pancake, Random Walks

Length	Solved	Cost	Optimal	Subopt.	Nodes generated
5	500	4.98	4.98	0.0%	89
10	392	9.21	9.19	0.2%	148,405
10	108	10.32	9.98	3.4%	11,150

24-Pancake, Random Walks

Length	Solved	Cost	Optimal	Subopt.	Nodes generated
5	500	4.98	4.98	0.0%	89
10	392	9.21	9.19	0.2%	148,405
10	108	10.32	9.98	3.4%	11,150
15	328	12.61	12.04	4.7%	109,535
15	152	14.94	14.05	6.3%	95,567

24-Pancake, Random Walks

Length	Solved	Cost	Optimal	Subopt.	Nodes generated
5	500	4.98	4.98	0.0%	89
10	392	9.21	9.19	0.2%	148,405
10	108	10.32	9.98	3.4%	11,150
15	328	12.61	12.04	4.7%	109,535
15	152	14.94	14.05	6.3%	95,567
20	416	15.92	14.77	7.8%	78,853
20	82	18.66	17.11	9.1%	124,345

24-Pancake, Bootstrap after RW

Iter	Solved	Cost	Optimal	Subopt.	Nodes generated
0	175	21.53	20.03	7.5%	267,476
1	1767	23.98	22.11	8.5%	238,351
2	1692	25.20	23.05	9.3%	262,412
3	781	25.77	23.36	10.3%	302,697
4	370	26.23	23.56	11.3%	284,175

How Effective are the Heuristics?

State Space	Avg. Solving Time 500 BS instances
35-pancake	158 secs
24-puzzle	273
20-blocks	2763
Rubik’s Cube	1 BS instance: 234

How Effective are the Heuristics?

State Space	Avg. Solving Time 500 BS instances	Avg. Solving Time 5000 BS instances
35-pancake	158 secs	4.4
24-puzzle	273	9.8
20-blocks	2763	23.0
Rubik's Cube	1 BS instance: 234	261.0

How Good are the Solutions?

State Space	Suboptimality 500 BS instances	Suboptimality 5000 BS instances
35-pancake	12.3%	17.6%
24-puzzle	6.1%	9.6%
20-blocks	3.6%	9.2%
Rubik’s Cube	1 BS instance: 20.5%	26.3%

How Long Does It Take?

State Space	Total Training Time 500 BS instances	Total Training Time 5000 BS instances
35-pancake	1 day	3 days
24-puzzle	2 days	18 days
20-blocks	2 days	11 days
Rubik's Cube	1 BS instance: 11 days	80 days

Solving Single Instances Quickly

Learning Time vs. Solving Time

- The Bootstrap method spends a large amount of time to create a heuristic ("learning time") that can then very quickly solve an instance ("solving time").
  - solving time << learning time
  - This is appropriate when the heuristic will be used to solve many instances
- A different balance between learning and solving times is needed if there is just one instance to solve.

Approach

- Interleave the learning process and the solving process, with a certain ratio of times for each (e.g. 5:1).
- No user-given instances other than the one to be solved. The fully automatic RandomWalk process used by Bootstrap is the source of training data. It iterates, creating more and more challenging instances, until the given instance is solved.

Approach (cont'd)

- What to do when a new heuristic is learned?
  - Kill the current solving process, start a new one with the new heuristic.
  - Spawn a new solving sub-thread, so that there is a solving sub-thread for every known heuristic.
    - How to allocate time among these sub-threads?
  - Keep running the solving process but replace the old heuristic with the new one. **(Heuristic Replacement)**

Heuristic Replacement Results

State Space	Total Training Time (Bootstrap-500)	Single Instance Avg. Total Time (suboptimality)
35-pancake	1 day (12.3%)	2h 34m (11.7%)
24-puzzle	2 days (6.1%)	**14m 28s (6.8%)**
20-		m (1.2%)
Rubik's Cube	11 days (20.5%)	10h 54m (5.1%)

Guaranteed optimal solutions are found in 12h 17m

A Different Approach

- Humphrey, Bramanti-Gregor, and Davis (1995)
- Bootstrap learning of a heuristic for a **single instance** based on **failed attempts** to solve it.

- BIG IDEA:
  - learn a function that estimates distance between any two states.
  - Each search generates lots of training data of this form, even if the problem instance is not solved.
- Impressive results on the 15-puzzle.

Solution Cost Prediction

- Bootstrap's main bottleneck is that it has no way of knowing which instances it can solve, so it wastes lots of time attempting to solve instances that can't be solved with the current heuristic.
- If we could accurately predict the optimal solution cost of any instance without actually having to solve the instance, we could avoid all of Bootstrap's failed attempts and its iterations.
- Solution cost prediction system: BiSS (Lelis et al.)

Which Training Instances?

- We are now free to use any set of instances we like to create a training set, so the focus of attention shifts from “how to get solution costs?” to “which instances should go into the training set?”.
- Prevailing wisdom is that a mixture of easy and hard instances are needed. Our approach:
  - Generate easy instances with random walks backwards from the goal.
  - Generate hard instances by randomly choosing a state.

Solving Single Instances

State Space	System	Total Time	Subopt.
24-Puzzle	BST	14m 28s	6.5
24-Puzzle	BiSS-h-500	9m 21s	5.7

Solving Single Instances

State Space	System	Total Time	Subopt.
24-Puzzle	BST	14m 28s	6.5
24-Puzzle	BiSS-h-500	9m 21s	5.7
35-Pancake	BST	2h 34m	4.6
35-Pancake	BiSS-h-500	30m 46s	4.7

Solving Single Instances

State Space	System	Total Time	Subopt.
24-Puzzle	BST	14m 28s	6.5
24-Puzzle	BiSS-h-500	9m 21s	5.7
35-Pancake	BST	1h 42m	4.6
35-Pancake	BiSS-h-500	30m 46s	4.7
Rubik’s Cube	BST	10h 54m	6.4
Rubik’s Cube	BiSS-h-500	1h 48m	10.4

Solving Single Instances

State Space	System	Total Time	Subopt.
24-Puzzle	BST	14m 28s	6.5
24-Puzzle	BiSS-h-500	9m 21s	5.7
35-Pancake	BST	1h 42m	4.6
35-Pancake	BiSS-h-500	30m 46s	4.7
Rubik’s Cube	BST	10h 54m	6.4
Rubik’s Cube	BiSS-h-500	1h 48m	10.4
Rubik’s Cube	BiSS-h-200	21m	11.5

20-Blocks World

- BiSS-h solved only 10 out of the 50 instances with a one-hour time limit per instance.
- Taking the training instances used by Bootstrap on its last iteration and predicting their solution costs with BiSS-h, BiSS-h outperforms Bootstrap.

	Learning Time	Nodes Generated	Subopt.
BST-5000 (13)	11d 1h	5,523,983	9.6
BiSS-h-2933	7h	279,513	8.8

Summary

- Long tradition of learning applied to planning/search.
- Lots of different ways that learning might be able to improve planning.
- Two main scenarios learning a heuristic function:
  - “unlimited” training time, expecting a very good, general heuristic at the end.
  - single problem instance, solve it as quickly as possible.
- Bootstrap does well in both scenarios.
- BiSS-h is even better, but there’s an open question about which instances it should label.
- **Contact info: rholte@ualberta.ca**