Automatic Domain Analysis

- Automatic analysis is only as useful as its potential exploitation.
- Domain engineers know more about domains than automatic analysis can use (automatic analysis will only ever have access to a subset of the domain engineer’s knowledge).
- Off-line analysis: analyse the domain before having access to a problem – like a compiler looking for code optimisations.
- On-line analysis: analyse the domain and the problem together – like an interpreter using run-time optimisations (Just-In-Time compiling).
- On-line analysis can be "Static (prior to planning) or Dynamic (during planning)."

A (far from complete) Catalogue of Domain Analyses

- Control rule learning [Minton, Etzioni]
- Macro-operator learning [Fikes and Nilsson, Korf]
- Generating abstractions [Kooblock]
- Inference of invariants [Morris and Feldman, Kelleher and Cohn, Blum and Furst, Gerevini and Schubert, Fox and Long, Rintanen, Scholz]
- Filtering domain and problem content [Nebel, Dimopolous and Koehler, Fox and Long]
- Heuristic evaluation function generation [McDermott, Bonet, Loerincs and Geffner, Refanidis and Vlahavas, Hoffmann]
- Planning strategy selection (based on domain characteristics) [Hoffm et al]
- Goal ordering [Koehler and Hoffmann, Porteous and Sebastia, Refanidis and Vlahavas]
- Identification of sub-problems [Fox and Long]

Learning control knowledge

- PRODIGY−EBL [Minton et al] - learns from examination of generated plans.
- But... Minton observes (AAAI96) that the learning process hard codes (typically domain-dependent) assumptions about the kinds of rules that are useful.
- Eg: PRODIGY−EBL examines a trace of a solution and identifies a goal satisfied and then undone, only to be resatisfied.
  - Suppose this is on(A,B) before on(B,C) in the blocks world.
  - PRODIGY learns that putting on(X,Y) before on(Y,Z) causes a goal interaction.
  - Now the fact is converted into a control rule (for the domain): if on(X,Y) and on(Y,Z) are goals, do on(Y,Z) first.
  - This control rule is based on a domain-dependent observation: goal interactions are generally best avoided.
- Minton proposes that learned rules must be subjected to utility evaluation.

What does it cost?

- Learning techniques are applied off-line.
- Learning requires previously generated plans (examples) and these can be expensive to obtain.
- Learning often involves "explaining" observed phenomena; phenomena must be found within examples and matched against structures of explanations. This is usually NP-hard.
- Checking utility is expensive – requires multiple planning attempts and post-planning analysis.
- The costs are high so no use for small numbers of real planning runs.
**Macro-operators: Learning procedural abstractions**

- A Macro-operator is a (usually learned) sequence of primitive operators that can be used as if it were a primitive operator to satisfy conjunctions of goals.
- Korf demonstrated the possibility of learning such sequences.
- They have particular utility in domains such as Rubik’s cube problem, in which satisfying goals involves local reversals.
- Other work on macro-operator sequences includes Porteous’ and McCluskey’s Propose system.
- Key problem:
  - Many operator sequences can be constructed;
  - Retrieval of relevant macro-operators can be as hard as construction [Nebel and Koehler].

**Automatic Generation of Abstract Operators**

- Knoblock (AI??) explored automatic construction of abstract operators from primitives.
- Abstraction based on relaxation of operators (removal of literals from pre- and post-conditions).
- Planning proceeds with abstracted operators and then literals are reintroduced in stages and plans extended to treat new literals (ALPINE system).
- Abstraction based on construction of dependency orderings between pre- and post-condition literals.
- Successful in Towers of Hanoi problems, where ordering leads to planning for successive discs, starting with the largest.
- Unfortunately, many (most?) problems have too little dependency-ordering to generate deep abstraction hierarchies.

**Automatic inference of invariants**

- Considerable work now exists on inference of invariants: DISCOPLAN [Gerevini and Schubert], TIM [Fox and Long] and systems by Rintanen and Scholz.
- Two broad approaches: Hypothesize-and-test and Analyse-and-generate.
- Hypothesize-and-test involves identifying plausible candidate invariants for a domain (described in an action-centric language such as PDDL), then testing them against the domain.
  - DISCOPLAN (generates a wide variety of invariants from a rich source language);
  - Scholz and Rintanen also adopt this approach.
- Analyse-and-generate involves constructing a model of the behaviors of objects in the domain and then generating invariants based on the model (from which the invariants are more readily accessible).
  - TIM constructs a FSM-based model of the domain.

**What use are invariants?**

- Some uses already explored:
  - Domain consistency checking;
  - Graphplan mutex relations [Blum and Furst, Fox and Long];
  - Simple control knowledge in SATPLAN-style planners [Kautz and Selman, Gerevini and Schubert];
  - Problem structuring and goal ordering [Refanidis and Vlahavas, Porteous and Sebastian].
- Other uses still relatively unexplored:
  - Improving heuristic function estimates;
  - Aid in restructuring sequential plans into parallel plans;
  - Further aid in problem structuring.
An example of invariant generation: DISCOPLAN

- Look for co-occurrence of a literal, A, as positive effect and a second literal, B, that is either an effect or a persistent precondition, with B containing a proper sub-set of variables from A.
- Hypothesize invariant $A \rightarrow B$ and test it against all actions.
- If the hypothesis fails, supplement the antecedent with "excuses" that weaken the invariant and retest.

Eg: $at(X,Y) \rightarrow \neg in(X,Z)$

Other invariants include single-valued constraints, anti-symmetry constraints, XOR constraints

An example of invariant generation: TIM

TIM constructs a FSM-based model of the behaviours of objects in the domain.

For example:

\[
\begin{align*}
&\text{at1} & \text{in1} \\
\text{load} & | & \text{unload}
\end{align*}
\]

Invariants:

\[
\forall x. (\exists y. at(x,y) \text{ or } \exists y. in(x,y))
\]
\[
\forall x. \neg (\exists y. at(x,y) \text{ and } \exists y. in(x,y))
\]
\[
\forall x,y,z. at(x,y) \text{ and } at(x,z) \rightarrow y = z
\]

Costs for Invariant Construction

- Hypothesize-and-test:
  - Costs depend on how good hypotheses are, how many there are and how many "excuses" are used to try to save a failed hypothesis.
  - In general an arbitrary cut-off is used to limit "excuse" generation.
- Analyse-and-generate:
  - Depends on the cost of model-construction and subsequent generation.
  - Typically limits the variety of forms of invariants that can be constructed.
- Existing implementations of both approaches are efficient, but generate only a subset of all possible invariants.

Filtering domains

- Domain encodings can contain redundant information (with respect to solving a specific problem).
- Problem initial states can contain redundant information (with respect to a specific goal).
- Filtering out useless domain and problem structure can dramatically reduce unnecessary search.
- What can be filtered?
  - Useless objects;
  - Useless operators;
  - Useless operator instantiations;
  - Useless symmetric choices.
Techniques for Domain Filtering

- Determining whether a domain component is irrelevant is, in general, as hard as planning.
- Nebel, Dimopoulos and Koehler (ECP97) present RIFO to heuristically filter irrelevant domain information.
  - Construct an and/or tree backwards from goals (or−nodes are alternative achieving action instances, and−nodes are required collections of goals and preconditions).
  - Heuristically minimise the set of initial facts required to support the goals.

RIFO Example

- Op1: Pre: \{a, b\}, Add: \{x\}, Del: \{a\}
- Op2: Pre: \{b, c\}, Add: \{x, y\}, Del: \{b, c\}
- Op3: Pre: \{a\}, Add: \{y\}, Del: \{a\}

![Diagram of RIFO example with nodes and actions]

Propagate possibilities upwards.
Keep only smallest \(n\).

More Techniques for Domain Filtering

- Some operator instantiations can be determined to be useless:
  - Fox and Long show that by identifying certain problem structures, some actions can be proved to be unhelpful and never instantiated.
  - Type information can be used to avoid operator instantiations that are necessarily redundant.
  - Gerevini and Schubert avoid operator instantiations using automatically inferred restricted parameter domains.

- Fox and Long have examined the automatic recognition of functionally equivalent objects (symmetric objects):
  - Equivalent choices (up to symmetric objects) can then be filtered out of a planner's search space.
Goal Ordering and Problem Decomposition

- A key choice point in many planning systems is which goal to tackle next. Making the right choice matters.
- Eg: goals on(a,b) and on(b,c) should be satisfied in the order on(b,c) then on(a,b).
- Several researchers have explored automatic ordering of the goals:
  - Koehler and Hoffmann order top-level goals (GAM);
  - Porteous and Sebastia extract a collection of landmark goals that must be achieved during the plan's lifetime and then order these.
- Identification of key intermediate states (or sets of alternative states) that a plan must visit allows a divide-and-conquer approach to planning.

GAM

- Two goals, g and h, should be ordered g < h if all plans for achieving g from a state in which h is true will force h to be undone.
- It is far too expensive to consider all plans.
- A heuristic approach is used in which the set of operators to be considered is reduced to a subset that cannot make h false, O_h.
- It is then confirmed that no plan built using these operators could make g true, from a state in which h is true.
- Assume (heuristically) nothing else may be considered true in start state.
- Ordered goals then collected into a totally ordered sequence of sets (ordered by inclusion) respecting the ordering between goal pairs.
- This sequence of sets is treated as an agenda to be followed in achieving the top-level goals.

GAM continued

- GAM relies on construction of a collection of facts inconsistent with a given goal.
  - Either use graphplan fix-point layer or...
  - ...cheaper, but less effective, direct analysis of operators achieving given goal.
- GAM uses an inductive process to confirm second goal unachievable using reduced operator set — reminiscent of hypothesize-and-test invariant generators.
- GAM concentrates on ordering top-level goals.

Alternative Goal Ordering

- Potorous and Sebastia have recently considered a different approach:
  - Identify landmarks — critical literals that must be made true on a route to the goals;
  - Order landmarks by examining relationship between possible achieving actions;
  - New orders arise when considering inconsistencies between side-effects of possible achievers on one goal and the possible truth of a second goal.
- This approach also requires knowledge of inconsistent facts.
- Uses TIM to determine inconsistent pairs efficiently.
- Refanidis and Vlahavas have also looked at problem structuring using invariants.
- Goal ordering is a powerful technique in strongly structured domains (blocks world), but not as powerful (yet?) in domains with several (possibly parallel) strands of activity.
Automatic Construction of Heuristic Evaluation Functions

- Approaches using abstracted forms of the planning problem used to evaluate alternative action choices:
  - UNPOP [McDermott];
  - HSP [Gelfond and Lifschitz];
  - GRT [Refanidis and Vlahavas];
  - FF [Hoffmann].

- Weak heuristics, but surprisingly effective.
- All examples demand construction of instantiated actions.
- Some ADL approaches use expansion into normal form, which is potentially exponentially expensive.

Domain and Problem Characterisation for Planning Strategy Selection

- Howe et al (ECP99) have explored statistical “fingerprinting” of domains and characteristics of planning strategies.
- Attempt to match domains and problems to planning strategies automatically, using simple measures of domain and problem characteristics, such as:
  - size of initial state;
  - number of objects in problem;
  - number of goals;
  - number of action schemas;
  - number of predicates.
- Multi-planning strategy system used AIPS98 competition planners (plus UCPOP and Prodigy) and time-shared resources according to expected behaviour on given problem.
- Improved robustness, reasonable time performance.

Automatically Identifying Sub-problems

- Knowledge-sparse planners need good control knowledge.
- A great deal of good control knowledge has been invested in solving specific hard problems (e.g., TSP, resource allocation, multi-processor scheduling, timetabling...).
- If a system can identify a known sub-problem then this control knowledge could be harnessed.

- A key development direction in research on TIM [Fox and Long] is the automatic recognition of sub-problem structures in planning problems.
- These sub-problems rest on generic types.
- A generic type is a collection of types of objects that share the same fundamental behaviour. Examples include:
  - Mobile objects;
  - Portable objects;
  - Construction materials;
  - Resources.

Fingerprinting Domain Structure

- Generic types can be identified with “fingerprints” characterised by specific structured and relationships in the domain model.
- These fingerprints can be found by examining the FSM structures TIM constructs to model a domain.
- Eg: A fingerprint for a mobile object that enters an intermediate state between locations. This can be used to model resource consumption (time or fuel) during movement.
What can we do with sub-problem identification?

- Specialised solvers are far more efficient than generic solvers.
- Integration of specialised solvers into a generic problem solving framework has been explored to some extent:
  - ItTeT [Ghallab et al.]
  - HSTS [Muscettola et al.]
  - O-Plan [Faite et al.]
  - Sipe 2 [Wilkins et al.]
  - Zeno [Weld and Penberthy]
  - TRP [Cesta and Stella]
  - STAN4 [Fox and Long]

Why automatic?

- Domain engineers don’t always see a domain the right way to recognise a particular sub-problem (the five-armed-aliens problem).
- Domain engineers sometimes propose sub-optimal control knowledge (the candle problem).
- Domain engineers shouldn’t have to know what a planner can exploit.
- Domain engineers shouldn’t have to state the obvious.
- Domain engineers might interpret control knowledge differently to the planner (what is a resource? When is it safe to force a compiler to inline a piece of code?).
- Portable domain descriptions shouldn’t contain planner-specific control knowledge.

Domain Analysis and Pre-processing:

Issues, thoughts and conclusions

- Tools to support domain engineering.
- Separation of control knowledge and domain knowledge.
- Automatic support for consistency checking in domain knowledge.
- Automatic support for control knowledge extraction.
- Exploitation of richer forms of control knowledge.
- Reuse of good solutions to hard problems.