Symbolic Planning after GraphPlan and SATPLAN

- Many subsequent algorithms borrowed ideas from GraphPlan and/or SATPLAN
- They didn't necessarily propositionalize everything; some added costs, probabilities, etc.
- They compiled and used state invariants; & often used a *priority queue of partial plans*; (e.g., that an object can be in only one place) into the constraints;
- E.g., Gerevini et al.'s LPG borrowed the idea of random search to try satisfy constraints (it won in the 2003 ICAPS competition)
- Some used distinctive new ideas, notably Helmert's "Fast Downward" planner (it won in the 2004 ICAPS competition)

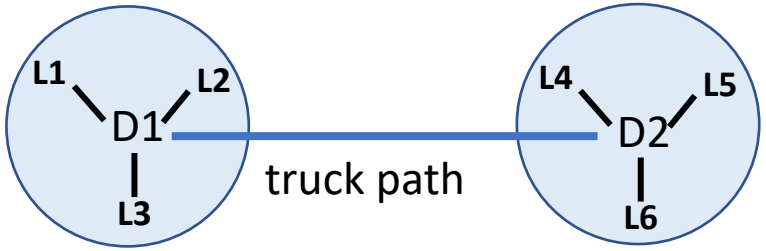
Most methods use The PDDL extension of STRIPS operators (Planning Domain Description Language)

Malte Helmert's Fast Downward Planner (https://planning.wiki/_citedpapers/planners/fd.pdf#page=1)

Novel ideas:

- Finds "multi-valued properties" of entities; e.g., a parcel can only be at one of L1, …, L6 or in one of the cars or in the truck; actions then just "set" values of variables, w/o deletions
- Planning in stages: infer that each parcel needs to undergo a load-unload-load-unload-load-unload sequence; choose vehicles for that, then fill in rest of plan;
- Finds what entities (e,g, cars, trucks) can "cause" other entities (e.g., parcels) to change properties; uses heuristic that adds up costs of changing properties to goal values, given available causes.

E.g., parcel delivery domain



Internal routes use cars, depot-to-depot uses truck

Tom Silver et al. "PDDL Planning with Pretrained Large Language Models", NeurIPS 2022

https://openreview.net/pdf?id=1QMMUB4zfl

Idea 1:

- Use Codex LLM, *few-shot prompting* with PDDL-encoded sample problems in each domain, obtained with Helmert's FD (e.g., two-armed robot moves balls from one place to another); a step-by-step variant checks each newly generated step for validity (preconds satisfied), and invalid step are replaced by the most similar valid one (with similarity measured on vector embeddings of the steps).
  Works poorly – success in 17% of examples with the one-shot method, 22% with step-by-step.

Idea 2:

- Use plans obtained as above (sometimes complete, sometimes partial) for stepwise guidance of the symbolic planner (FD), speeding it up; (solutions are guaranteed correct for FD planner) do this by adding all prefixes of the LLM's plan the priority queue; then, at each new step, make the choice as usual using the estimated-cost-to-completion heuristic.
- Ususally this gives reductions in partial plans created, 14% - 99.96% (the latter, for domains where the LLM plan is almost always correct). But sometime 3 X as many are created

Vishal Pallagani et al., "Plansformer: Generating Symbolic Plans using Transformers", Dec 16, 2022.
arXiv:2212.08681v1 [cs.AI],
https://arxiv.org/ftp/arxiv/papers/2212/2212.08681.pdf

Main ideas:
- They also use Helmert's FD planner and Codex, but rather than using few-shot prompting in each domain, they generating up to 14,000 examples per domain for fine-tuning the LLM and testing;
- They get about 90% valid plans that way (in domains hanoi, grippers -- moving balls across rooms, and driverlog -- the parcel delivery domain);
- They use "heavy machinery": 9 (or is it 9+44?) GPU nodes, 24 cores & 128 GB of RAM each, and Big Data nodes with 1.5TB of RAM, with access to 1.4 PB of GPFS (?) storage.

But with all the domain-specific fine tuning (taking 15-45min each) this doesn't seem very exciting.
The FD planner by itself still works better by itself.

Iddo Drori et al., "A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level", PNAS2022 Vol. 119 No. 32 e2123433119;

https://doi.org/10.1073/pnas.2123433119

Main ideas:
- They use few-shot prompting with examples of various math problems, showing how to solve them using NumPy and SymPy python libraries and sublibraries to write code to generate solutions.
- Then they ask, "Using numpy, <*original unaltered problem statement*>", e.g., *find the solution of …*

But basically, it's the NumPy/SymPY library code that's solving the problems, giving 71-80% success!