PUSH THE BOX

By oded fried

Using graph plan to solve puzzle games
1. The Game

The rules:

Push The Box is a one player puzzle game.

The goal is to get all the boxes: to the targets: . This is done using the orange machines: from now on we’ll call these machines “pushers”.

The player can open and close the pushers. When opening, the pusher will push anything in its way until it will hit a wall or another object.

“Free” or “empty” tiles appear green: “walls” are represented by bricks: 

The twist in Push the Box is, that pushers can also be pushed. In order to achieve the goal, the player must perform a series of pushes to align the objects on the board in a way that the boxes can be pushed to the targets without obstacles in their way.

Why is this an interesting problem?

This seemingly simple game has very simple rules, but can create a very difficult problem. In the original game, a small board of size 6x8 can hold a problem that requires over 40 moves to solve. As we’ll see later on, as solution length, border size and number of pusher’s increases, the problem becomes exponentially harder.

2. Solving the game using AI techniques

I decided to solve the problem with graph plan and compare it to a search problem approach

A search through state space:

Defining the game as a search problem is very strait forward.

States are any combination of walls, free slots, boxes and pushers (open or closed), and a set of non-wall tiles that are the targets. Actions are the activation of pushers, in every state there is one action for every pusher. For a given state and pusher to activate, the state transition function will be, closing the given pusher (if open) or opening and pushing an object according to the given state (if closed). An initial state any state and a goal, a state in which all the target tiles has boxes in them. As we did in the course, I limited myself the assumption that a solution exists.
Implementing basic search agents

In order to solve the problem I implemented generic DFS, BFS and A* search agents that can solve every state transition search problem. We “fed” a provided game logic unit to supply the state transition function, these agents perform a search through the game’s possible states looking for the goal.

A* heuristics

As I thought before starting the project, finding A* heuristics for the game was very hard. I first implemented the basic target count heuristic. This heuristic is simply calculated by counting the “unfulfilled targets” and targets that occupy pushers. The reason, of course, is that for every empty target we will need at least one move to push a box to it, and another move to remove the pusher (because every move can move at most one object).

A more advanced heuristic is the stuck box heuristic. This is a method that identify “dead ends” in an early stage. This method looks for objects that can’t move anymore and returns infinity as a value for states that have boxes that are not on targets, and can’t move anymore. For other non-goal states, the return value is one. It is interesting to see how a heuristic function that returns only 0, 1 and infinity can prune significant portions of the search tree.

As we’ll see in the next section, my project is all about plan space, I used the graph plan algorithm to provide a relaxed plan heuristic. The plan graph is being expended without computing mutually exclusive relations to provide a relaxed plan. The function returns the number of the first layer that contained the goal.

All three heuristics are admissible. (Target count, as already explained, will always return a number smaller or equal to the true minimal number of steps towards the goal. Stuck box is admissible because when a box is stuck, there is no solution and the distance to the goal can be considered as infinity. The relaxed plan will have at least one new action that will bring us to the goal, otherwise the search would have ended in this layer).

Only stuck box is consistent, which is trivial due to the few return values (0, 1, in). Target count can be “fooled” by boxes in targets that won’t be their final tiles and therefore not consistent. Relaxed plan is can give low values to state far from the goal if it can be achieved by several independent actions, and give high values for states that has a sequential path to the goal. In this specific realm.

I add another semi-heuristic called combined heuristic. This is a method of combining multiple heuristics to one dominating function by taking maximum h value of the state over all the function. Since all the supplied heuristics are admissible, combined heuristic is also admissible.

Note that while the relaxed plan heuristic dominates the stuck box heuristic (if there is no solution, relaxed plan will reached a fixed point with no goal and return infinity). Combining them is still helpful. That’s because relaxed plan is relatively “expensive” to calculate, and stuck box find and eliminates exactly the states that are most expansive for relaxed plan.
Why choose graph plan?

Thinking about the problem, I recognized a certain property of this specific problem:

A problem will usually have many solutions, which are all just one path to the goal only with different order.

This quality gives a partially ordered plan an advantage.

A problem with a solution of length 14, and only 5 pushers will have $5^{14} = 6,103,515,625$ states to go through. Which makes solving an “advanced” problem take extremely long time, and very memory demanding (for BFS and A*).

To be honest, my first intention was to write a POP algorithm that will search backwards from the goal using partially ordered plans as states. When I approached the problem more deeply, I realized that in the Push The Box Domain the branching factor for backwards planning is much larger than for forward planning (a state will have more possible predecessors than successors). Taking this fact into consideration, I decided to go with a forward expanding graph plan, and a backwards plan extraction (which is easier the backward planning because that way I know who are the reachable predecessors of the goal.

Representing the problem in STRIPS

A state will be described by the following propositions:

a. Types of tiles:
   at(x,y)-wall (indicates a real wall or an open pusher)
   at(x,y)-free
   at(x,y)-object (indicates boxes and closed pushers)

b. Objects:
   at(x,y)-box
   at(x,y)-pusher-closed-d (d indicates the pusher’s direction)
   at(x,y)-pusher-open-d-reach-r (r indicates how many tiles the open pusher takes)

Notice that this is not the most compact way to describe the state (we use two proposition for every object), but it will make the actions a lot simpler.

Also notice, that targets aren’t expressed explicitly, they will be expressed through the goal set.

The goal $G$ will be:

$G = \{at(x,y)\text{-box} \mid (x, y) \text{ is a target}\}$

The actions will be described in three groups:

a. Open-pusher-at(x,y)-reach-r

   $\text{PRE: } \{at(x,y)\text{-pusher-closed-d}\} \cup \{at(x+i,y+j)\text{-free} \mid 0<i<d_x, 0<j<d_y\} \cup$
   \{at(x+d_x,r,y+d_y)\text{-wall}\}

   $\text{ADD: } \{at(x,y)\text{-pusher-open-d-reach-r}\} \cup \{at(x+i,y+j)\text{-wall} \mid 0\leq i < d_x, 0\leq j < d_y\}$

   $\text{DEL: } \text{PRE} \cup \{at(x,y)\text{-object}\} \setminus \{at(x+d_x,r,y+d_y)\text{-wall}\}$

   ($d_x$ and $d_y$ are in $\{-1,0,1\}$ and are inferred from the direction of the pusher)
b. Pusher-d-at(x_p,y_p)-push-object-o-at(x_o,y_o)-reach-r-on-type-t
   (t stands for object/wall, o stands for the type of object being pushed)
   **PRE**: {at(x_p,y_p)-pusher-closed-d} U {at(x+p,d_x*r,y+d_y*r)-t} U {at(x_o,y_o)-o} U {at(x_p+i,y_p+j)-free | 0<i<d_x*r, 0<j<d_y*r, x_p+i≠ x_o, y_p+j ≠ y_o }
   **ADD**: {at(x_p,y_p)-pusher-open-d-reach-r} U {at(x_p+r,y_p+r)-o} U {at(x_p+i,y_p+j)-wall | 0<i<d_x*r, 0<j<d_y*r }
   **DEL**: PRE U {at(x_p,y_p)-object} \ {at(x_p+r,y_p+r)-o} 
   (d_x and d_y are in {-1,0,1} and are inferred from the direction of the pusher)

c. Close-Pusher-d-at(x,y)-reach-r (here r is the old reach)
   **PRE**: {at(x,y)-pusher-open-reach-r} U {at(x+i,y+j)-wall | 0<i<d_x*r, 0<j<d_y*r }
   **ADD**: {at(x,y)-pusher-closed-d} U {at(x+i,y+j)-free | 0<i<d_x*r, 0<j<d_y*r } U {at(x,y)-object}
   **DEL**: PRE

Of course, for the graph plan, we also an NO-OP actions.

**Implementing the graph plan agent**

I tried to make the graph plan agent as generic as possible, except for the proposition layer, the graph plan agent can solve any given STRIPS problem using the layer expansion and gp-search algorithm we learned in class. The graph is initialized with layer 0, which is made up from all the initial state’s propositions. Then the graph is expended, while updating mutex relations between actions and propositions. When a proposition layer that contains the goal is reached, a recursive backtrack is performed to find a set of non-mutex actions that will provide the needed propositions. If the search fails, the set of propositions is added to a “no-goods” set, to indicate it can’t be reached in this level, this saves recurring computation as we might run into the same set of propositions on this layer in a different search.

**Optimization methods applied in the algorithm, both new, and ones we learned in class**

a. The proposition layer as is specialized for the Push the Box game. It is the only non-generic component. In this layer, the proposition are saved in a data structure that resembles the state’s structure (two dimensional array of lists). This structure enable’s easy access to proposition by their (x,y) coordination. But mostly, it uses the games special mechanics to generate actions by looking for “pusher propositions” which are essential for every action. This is in contrast to the naïve way of generating actions: for every action A, check if A’s precondition are in the layer, if so, add A to the new actions.

b. In the action layer, actions pre,add,del sets are computed during the search instead of in advance. This is to compute actions only on a need basis. Because of the nature of the game, there tend to be a lot more “possible” actions than actually reachable in a given Push the Box problem. Every action is computed once and saved in a set.

c. Another optimization I used, is remove impossible. That means, that we add actions that have all non-mutex preconditions. By doing that, we don’t add actions to the layer that can’t be reached there at all – and thus impossible.

d. Out of the three reason two actions can me mutually exclusive are interference and inconsistent effects. Those relations are unrelated to the layer, and computed only once for each pair, on a need only basis. (And saved in a
dependent/independent set

Note that this optimization, although related to mutex relationships, is applied also in the relaxed plan heuristic for almost no cost in computation time.

3. The GUI

Using the GUI

The game’s interface lets the user choose from a number of levels in different difficulties. Before the game starts, we can choose a “player” to play the game. Choosing “Human” will let the user try and solve the game by himself. The other players are: “DFS agent”, “BFS agent”, “A* agent” and “Graph Plan Agent”. When choosing either one of these players, and pressing the start button, the agent will start working on a plan, when it is done, the agent will inform us about his performance through the “chat” window and will start giving instructions in the chat and by marking the next tile to play in red.

The A* agent also has a “heuristics panel” that lets the user choose the desired heuristics using check boxes. (Choosing more than one heuristic will combine them with the “combined heuristic” that was explained earlier. Choosing none will load the agent with a null heuristic, and that will make it act as if it was a BFS agent)

The layout

1. Load level – choose the layout of the level. Push the box layout files are marked with the suffix .pbl
   the levels are ordered by difficulty: beginner, novice, advanced, skilled and expert.
2. Reset level/start – the reset level button will restart the same level. The start button will trigger the agent’s planning. During his time the program is unresponsive.
3. Player chooser – choose who plays the game
4. Level information – the level’s name, length of the optimal solution and number of moves already made.
5. The chat – here the agent or the game will tell us what they are thinking or if something went wrong.
6. The agent marks the tile he wants to play next

Comments

The GUI and the games interfaced are written in a very generic way. The game’s interface supports multiple players and the GUI can work with every game that works on a grid.
4. Testing the agents:

I tested the different agents on levels of varying difficulty from the smartphone game. Some results were surprising and others were expected. The first noticeable thing is the exponential growth of computation time. Most agent did very well on small level (plan under one second) but failed terribly on levels a little larger (out of memory / time out after an hour).

*A* heuristics

The three heuristics were examined in the following manner: the A* agent played on four times on every board, one for every heuristic, and one time using all of them. I compared the time it took to the agent to find the solution and the number of nodes expanded.

A measurement I thought was helpful was the ratio between the number of nodes the heuristic expanded and those the BFS agent expanded (the BFS acts a null heuristic testing group)

The relaxed plan heuristic showed excellent results. With number of nodes usually around ten to a hundred times less nodes that the other heuristics. As levels became harder, this ratio grew even more. This heuristic did pretty well on all levels tested and maintained a small numbers of nodes while the other heuristics grew. In the other hand, the relaxed plan heuristic was a much more time consuming then its counterparts. The average nodes per second for the tree heuristic is. 113 for the relaxed plan, 613 for target count, and 633 for stuck box (nodes per second). And therefore, in most of the time the other two heuristic did better time than relaxed plan.

I would still choose relaxed plan as the better heuristic, because it showed the tendency to keep search tree minimal and I would suspect that with higher performance machines, and while dealing with larger scale problems, it would have worked better than the others.

There wasn’t an apparent better between the heuristics target count and stuck box. They both did well on some levels and les well on the others. In levels that holds few boxes, target count heuristic can’t contribute too much, on the other hand, and there are some levels that boxes can’t get stuck at all.

We can see, especially in advanced levels, that when we combine the three heuristics, relaxed plan becomes the dominant one, that’s because it is very time consuming, and it’s impact on the node count is huge. We already showed that it dominates the stuck box heuristic. That said, in some levels we can see similar node count in the combine and relaxed plan, but we see an improvement in time because the faster heuristic saves us computation time on easy to find, dead ends.

The underdog

The DFS agent, was put on the project mainly for contrast in the different behavior of agents, and because it was easy to add it to the generic code. To my very surprise, this agent showed his unique and interesting look on the game. On almost every level, no matter how hard, and no matter how many other agents died of lack of memory trying to solve it, DFS agent found a solution in an extremely low amount of time. On some levels it is the only agent that could come up with any solution, Of course, while not considering optimality at all, the solution was sometimes horribly large (hundreds of turn for an optimal solution of length 20). Maybe the solution found by DFS can be simplified to provide a valid solution in a small amount of time.
**Graph Plan agent**

The main part of this project was this agent. While showing a slower start than the search agents, the graph plan agent had the most success of all the agents.

This agent was not faster than the other agents on most levels, but could solve puzzles that the others could not. The upper limit for the A* agents was around 18 moves of solution length, beyond that they mostly get MEMOUT or they don’t solve the game in a very long time.

The graph plan agent is not optimal. As it will sometime choose more than one action in a layer that achieve its goal and doesn’t check if it could be done in fewer actions. In this particular game it shows when it opens and closes pusher for no apparent reason.

The truth is, it sometimes try to make a tile free by having an open pusher it can close there.

The areas this agent performs exceptionally well are were parallel planning is available. I created special levels that several small ones together. On those levels, while the solution length climbed over 20, the agent’s way of solving the problem made the difference between the separate parts of the problem. This cannot be done while planning in state space because then, when choosing states we also choose their order and can’t tell if there is a difference or not. The best demonstration is the incredibly simple for humans, but hell for plan agents called many boxes: (can you solve it?)

![Image of a puzzle](image)

Well... most of the agents can’t! None of the three heuristics work here and only the graph plan agent solves it (no problem).

The only intelligent thing to do is to realize that the boxes and pushers are independent of each other and the ordering does not matter.

**What’s next?**

While Building the graph plan agent from scratch, there were many ideas that came to mind wish I had more time to try and implement. Like implementing heuristics for the plan extraction (like choosing actions before no ops to run into dead ends quicker, or smallest set of action possible at each layer to get an optimized solution).

I would like to implement a graph plan agent that doesn’t uses search backwards tracking but a more optimized CSP solver that looks for things in between the layers (like a proposition that has only one provider etc.)