TASK PLANNING IN FACTOR GRAPH REPRESENTATION

A Dissertation
Presented to
The Academic Faculty

By

Yoonwoo Kim

In Partial Fulfillment
of the Requirements for the Degree
Masters in the
College of Computing

Georgia Institute of Technology

May 2023

TASK PLANNING IN FACTOR GRAPH REPRESENTATION

Date approved: January 13, 2023

I dedicate this thesis to,

my father Byung Cheol **Kim**,

my mother Miran **Yoon**,

and my brother Do**woo** Kim.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my advisor Prof. **Frank Dellaert**. Frank introduced me to research and taught me skills like organizing and reading papers, test-driven coding style, and delivering scholarly work.

I would also like to thank my thesis committee members for their help in preparing this thesis. Comments from Prof. **Matthew Gombolay** and Prof. **Danfei Xu** were immensely helpful in improving the quality of this thesis.

I am also grateful to **Taery Kim** for helping organize the experiments and being unhesitant in giving advice on research. Thank you, **John Yi**, I really enjoyed our late-night workout sessions at the CRC. **David**, **Zachery**, **Hansung**, **Dowon**, **Sean**, **Beom**, **Soohyun** and **Kyumin**, thank you all for making my stay at Georgia Tech enjoyable and vibrant since undergrad.

Finally, I would like to thank my family, **Byung Cheol Kim** (father), **Miran Yoon** (mother), and **Dowoo Kim** (brother), for their unconditional support throughout this journey.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

Many optimization problems in robotics are known to be local, where the problem depends only on a subset of variables (Dellaert 2021). For example, in classical planning, an action chosen between time steps 0 and 1 depends on variables in those time steps, not variables from time step 2.

A factor graph is a probabilistic graphical model that has its strength in exploiting the locality property of an optimization problem. Due to this advantage, it is used to model various problems across robotics, such as simultaneous localization and mapping (Dellaert et al. 2017a), structure from motion (Baid et al. 2021), motion planning (Mukadam et al. 2018), etc. Furthermore, the generality of the factor graph enables a unified probabilistic framework, such as simultaneous trajectory estimation and planning (Mukadam et al. 2019).

Inspired by this generality, this thesis uses factor graph representation to model classical planning. The classical planning problem represented in a factor graph can be solved using maximum a posterior (MAP) inference, which maximizes the product of all factors in the factor graph. We propose a formulation where the MAP inference will output a series of concurrent actions with the highest probability of reaching the goal state. Classical planning is deterministic, so the probability of reaching a particular state is either True or False. Still, because the proposed method uses a probabilistic graphical model, it can handle probabilistic problems with partial observability and probabilistic success rate of actions.

However, MAP inference in discrete space can be computationally intractable because the discrete state spaces grow exponentially with respect to the number of discrete variables. This paper tackles this challenge by utilizing the sparsity of discrete state spaces. For example, most states in classical planning problems would be 0 due to predefined preconditions and effects. The proposed algorithm bypasses states with values of 0s while calculating the maximum a posteriori, significantly improving the computation time for sparse state spaces without any downfall, even in dense state spaces.

Finally, the proposed formulation of classical planning with a factor optimized for sparsity is tested on environments where concurrent plans are beneficial. In addition to existing domains for benchmarking, we create a centralized multi-robot environment where concurrent actions can enable shorter horizon plans.

Furthermore, the paper discusses the possibility of combining motion planning and classical planning with factor graph representation to create a task and motion planner. The formulation will form a discrete-continuous optimization problem where classical and motion planning problem is interleaved.

CHAPTER 1
INTRODUCTION

1.1 Motivation

Factor graphs can represent various problems across robotics (Dellaert 2021). Previous works, such as simultaneous localization and mapping (Dellaert et al. 2017a), tracking (Pöschmann et al. 2020), motion planning (Mukadam et al. 2018), and structure from motion (Baid et al. 2021), all use factor graphs as the underlying computing paradigm.

The generality of factor graphs suggests they can serve as a unified probabilistic framework combining different robotics problems. In fact, Mukadam et al. used factor graphs to simultaneously solve motion planning and trajectory estimation, leading to improved computation efficiency and accuracy in continuous motion plans. Yet, for robots to operate in an unstructured environment, they should be able to plan in both discrete and continuous space (Garrett et al. 2021).

Planning in discrete space is also known as classical planning. Classical planning in factor graph representation can be seen as finding an assignment to a set of discrete variables that minimizes the plan length while satisfying all factors. Still, classical planning in factor graph representation has not been thoroughly explored because the exponential increase in discrete state space with respect to the number of discrete variables makes inference on factor graph exponentially slow. Hence, this thesis explores how factor graphs can represent classical planning problems and solve them using off-the-shelf solver GTSAM (Dellaert et al. 2017b).

1.2 Contribution

To expand the use of factor graphs in robotics problems, our first contribution is **representing the classical planning problem in factor graphs**. Our representation enables combining task planning with various robotics problems, such as motion planning, on a single framework. This opens up new approaches for problems like task and motion planning on a factor graph representation which can be solved by discrete-continuous optimization methods (Doherty et al. 2022; Hsiao et al. 2019). Moreover, the representation naturally exposes actions that can be executed in parallel, allowing concurrent plans to be found. We test "classical planning using factor graphs" on scenarios that best exploit the advantages of concurrent plans, such as the gripper domain from IPC 1998 and the centralized multi-robot planning environment in Chapter 5.

Yet, the inference speed was too slow due to the exponential blow-up of discrete state space. Therefore, we made our second contribution, **developing discrete probabilistic factor optimized for sparsity**, which we call TableFactor. This factor ignores invalid states, for example, states that do not satisfy preconditions or effects, during computation reducing the execution time of factor multiplication by 19 times on average for sparse factors. We benchmark TableFactor against discrete probabilistic factor implemented in GTSAM (Dellaert et al. 2017b) varying the size and sparsity of the factors in Section 4.5. Classical planning problem with a horizon of 7 and $5^8 \times 3$ possible combination of actions per timestep could be solved in 0.96 seconds which is reported in Chapter 5.

CHAPTER 2
RELATED WORK

2.1 Overview

The related work chapter can be organized into four categories:

- **Factor Graph.**

  Since this thesis focuses on representing classical planning problems in factor graphs, we briefly explain what a factor graph is and how it is used to solve other robotics problems in Section 2.2.

- **Formulations of classical planning.**

  To represent classical planning problems in factor graphs, it is important to know how these problems are formulated. Therefore, we discuss two formulations used in the literature to formulate classical planning problems in Section 2.3 and examine which formulation is more suitable for factor graph representation.

- **Methods to solve classical planning problems.**

  There are multiple ways to solve classical planning problems, from forward search with heuristics to regression methods. We explain what classical planning is in Section 2.5 and show an approach from the literature closest to our approach to solving the problems in Section 2.6.

- **Extension of classical planning problems.**

  Task and motion planners often rely on classical planners to solve the “task” planning part of the problem. We include task and motion planning in Section 2.7 to illustrate how the proposed method in the thesis can be extended to solve a topic actively being researched in robotics.

2.2 Factor Graph

The approach proposed in this thesis uses a factor graph to represent classical planning problems. A factor graph is a bipartite graph with two types of nodes: variable V and factor F. The variable nodes are represented as circular nodes and factor nodes are represented as square nodes in graphical notation as shown in Figure 2.1. Each variable $v_i \in V$ can take on values in its domain $d_i \in D$, and each factor $f_i \in F$ takes an assignment of connected variables and returns a non-negative number representing how "good" that assignment is. In classical planning, this non-negative number is either 1 or 0 due to the deterministic characteristic of the problem.

A key aspect that makes factor graphs a powerful tool is their strength in exploiting the locality of the problem. That is, each factor f_i only depends on a subset of variables. Due to this strength, the factor graph has been used to model a wide range of robotics problems such as simultaneous localization and mapping, structure from motion, motion planning, etc. (Dellaert 2021). This thesis tries to broaden this spectrum by modeling classical planning problems in a factor graph.

2.3 Classical Planning Formulation

2.3.1 STRIPS Formulation

Stanford Research Institute Problem Solver, in short, STRIPS, is the root of our work as it was one of the first works to apply classical planning to robotics problems.

STRIPS was first proposed by Fikes et al. as a basic planning mechanism for the project "Shakey, the Robot" (Nilsson et al. 1984). A STRIPS formulation can be defined as a tuple $\Psi = (P, A, I, G)$ where:

- P is a set of propositional variables.
- A is a set of actions (or operators) where each action $a \in A$ is a form of $a =$

$(pre(a), add(a), del(a))$. $pre(a) \subseteq P$ is a set of preconditions, $add(a) \subseteq P$, and $del(a) \subseteq P$ are sets of add, and delete variables respectively.

- $I \subseteq P$ is the initial state.
- $G \subseteq P$ is a set of goal variables.

Following the STRIPS formulation, action $a \in A$ can be executed only if the current state has $pre(a)$ and will result in the addition of $add(a)$ and removal of $del(a)$. When representing STRIPS formulation in a factor graph, all propositional variables $p \in P$ will become binary true/false variable nodes. Using binary variables exponentially increases the state space with respect to the number of propositional variables p, making the formulation unsuitable for translating into factor graphs. Precisely, when translated to a factor graph, n propositional variables have the same expressivity as a multi-valued variable with a cardinality of n, but n propositional variables have a state space of 2^n whereas one multi-valued variable has a state space of n. For this reason, we use SAS+ formulation, which uses multi-valued variables to formulate classical planning problems further explained in the following Subsection 2.3.2.

2.3.2 SAS+ Formulation

SAS+ formulation was introduced by Bäckström et al. Instead of propositional variables that STRIPS formulation use, SAS+ formulation uses multi-valued variables.

Aside from the suitability of the formulation when translated to factor graphs, SAS+ formulation can naturally describe "invariants", which is not explicitly recognized by STRIPS formulation. For example, suppose a ball can exist in n different possible locations at any given timestep. STRIPS formulation will have n propositional variables that indicate all possible locations on which the ball can be located. In this example, the "invariant" is the fact that the ball cannot exist in multiple locations at the same time. This means in STRIPS formulation, exactly one out of n propositions should be true. Such information is naturally embedded into SAS+ formulation, whereas in STRIPS, it is not.

Also, SAS+ formalism provides structures such as Domain Transition Graphs and Causal Graphs (Jonsson et al. 1998) that capture legal transitions of variables from $state_t$ to $state_{t+1}$ and causal dependencies. It is shown that leveraging such structures can lead to effective heuristics to solve classical planning problems in the state-of-the-art classical planner, fast downward planning system (Helmert 2006).

Similar to STRIPS formulation, SAS+ formulation can also be defined as a tuple $\Pi = (X, O, I, G)$ where:

- X is a set of multi-valued variables where each $x \in X$ has a set of possible finite values $dom(x)$.

- O is a set of actions (or operators) where each action $o \in O$ is a tuple $o = (pre(o), eff(o))$. $pre(o) \subseteq X$ and $eff(o) \subseteq X$ are sets of multi-valued variables.

- I is the initial state.

- G is a set of goal variables.

2.4 Planning Domain Definition Language

Planning Domain Definition Language (PDDL) is a standard encoding language for classical planning problems introduced in the 1998 International Conference on Automated Planning and Scheduling(ICAPS) (Aeronautiques et al. 1998). After its first introduction, a series of improvements were made with extensions. PDDL 2.1 (Fox et al. 2003) allows for expressing temporal domains, PDDL 2.2 (Edelkamp et al. 2004) adds derived predicates, and PDDL 3.0 (Gerevini et al. 2005) introduces soft constraints.

Planning tasks written in PDDL are separated into a **domain file** and a **problem file**. Domain file establishes the context of the environment with a series of predicates and possible actions. Predicates are properties of the environment in $true/false$, and actions are possible ways to change the environment. An example of a domain file is in Listing 3.1.

A problem file represents an instance of the environment by defining what is true at the initial state and what we want to be true at the goal state, along with a list of objects that the planner should consider. An example of a problem file is in Listing 3.2.

2.5 Classical Planning

This thesis proposes methods to represent classical planning problems as factor graphs. Classical planning problem which is the problem of finding a sequence of operators from an initial state to a goal state in a fully observable and deterministic environment. An operator is a function from state X_t to X_{t+1} defined by precondition and effect. The precondition of an operator specifies when the action is legal in state X_t, and the effect of an operator specifies the resulting state X_{t+1}. Due to the precondition and effect, not every operator is legal between states X_t to X_{t+1}.

Formally, classical planning formulated in SAS+ uses a set of multi-valued variables X as a state and O as a set of possible operators per timestep. The goal is the find a sequence of operators $\pi =< o_1, o_2, \ldots, o_n >$ where $o_1, o_2, \ldots, o_n \in O$ that induces a sequence of states $< X_0, X_1, \ldots, X_n >$ such that $X_0 = I$, $X_n = G$ and all operators in π meets the preconditions and effects.

2.6 Planning as Constraint Satisfaction Problem

Planning as a constraint satisfaction problem (CSP) comes closest to our approach to solving classical planning problems. In fact, a CSP is a factor graph where all factors are constraints.

A constraint satisfaction problem consists of the following:

- $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables.
- $D = \{d_1, d_2, \ldots, d_n\}$ is a set of domains where $d_i(x_i)$ is a finite set of elements that can be assigned to x_i where $0 \leq i \leq n$.

- $C = \{c_1, c_2, \ldots, c_k\}$ is a set of constraints.

The constraint satisfaction problem aims to find assignments for all values in V that satisfy all constraints C defined in the problem. CSP solvers use a propagation algorithm during the search to filter inconsistent assignments and table constraints (Gent et al. 2007) to store a valid combination of assignments for a set of explicitly specified variables. Specifically, (Vidal et al. 2006) demonstrates that through careful design of models, it is possible to solve a significant portion of classical planning benchmarks through the propagation of constraints. We take a similar approach to the CSP solvers, with factor graph representation and `TableFactor` explained in Chapter 3 and Chapter 4.

Constraint satisfaction problem can model classical planning problem when the plan length is specified. Each operator's precondition and effect are converted to constraints along with the initial and goal states. The objective would then be to find an assignment for all variables which satisfies the aforementioned constraints.

Work by Barták et al. comes close to our method by reformulating CSP-PLAN (Lopez et al. 2003) previously written propositional variables to multi-valued variables from SAS+ using precondition and successor constraints. The precondition constraint is defined as:

$$O^t = o \rightarrow Pre(o)^t, \forall o \in Dom(O^t). \tag{2.1}$$

Equation 2.1 is a constraint that tells which actions are legal at time t. The successor constraint is defined as:

$${x_i}^t = val \iff O^{t-1} \in C(i, val) \vee ({x_i}^{t-1} \wedge O^{t-1} \in N(i)), \tag{2.2}$$

where $C(i, val)$ are operators which contains $v_i = val$ as their effects and $N(i)$ are set of operators which do not have state variable x_i among its effects. Figure 2.1 is a visualization of an example of the reformulated CSP-PLAN in a factor graph. Although the work (Barták et al. 2008) does not explicitly mention its relevancy with factor graphs, the fact that it can easily be represented with factor graphs shows the applicability of the graphical inference model in CSP planning.

We further extend this approach by directly representing the classical planning problem in factor graphs and formulating it to handle concurrent actions.

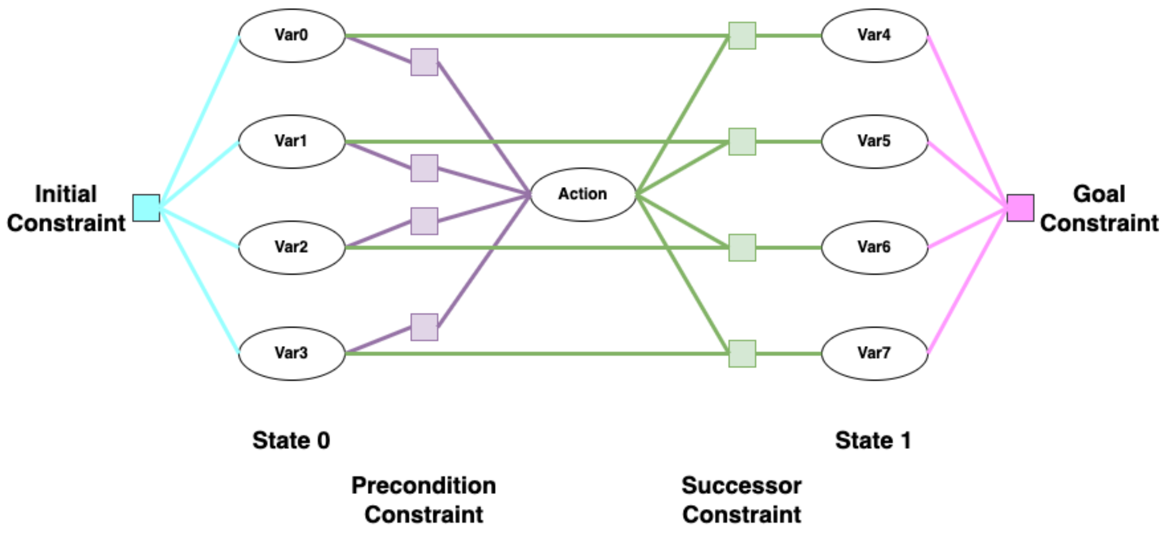


Figure 2.1: Example of reformulated CSP-PLAN represented in a factor graph for a plan with one horizon. Different colors indicate different types of constraints.

2.7 Task and Motion Planning

This thesis proposes a method to solve classical planning, which is the "task" part of task and motion planning (TAMP). Also, due to the wide applicability of factor graph representation, the proposed method can be integrated into motion planners using factor graph representation (Mukadam et al. 2018) to create a task and motion planner.

TAMP integrates discrete task planning from the AI field and continuous motion planning from the robotics field to allow robots to reason about high-level symbolic descriptions of the task and a lower-level geometric description of the environment during long-horizon planning.

TAMP problem can be defined using classical planning representation with preconditions and effects combined with constraints for continuous configurations of robots and objects in the environment (Garrett et al. 2021). The following is an example of Garrett et al.'s formulation for how actions can be defined in gripper domain:

moveF $(q, \tau, q', p^A, ...p^E)$

con: Motion(q, τ, q'), CFreeW(τ), CFreeA(p^A, τ)...CFreeE(p^E, τ)

pre: holding = None, atRob $= q$, atA $= p^A$, ... atE $= p^E$

eff: atRob $\leftarrow q'$

moveH[obj] $(g, q, \tau, q', p^A, ...p^E)$

con: Motion(q, τ, q'), CFreeW[obj](g, τ), CFreeA[obj](g, p^A, τ)...CFreeE[obj](g, p^E, τ)

pre: holding = obj, atRob $= q$, at[obj] $= g$, atA $= p^A$, ... atE $= p^E$

eff: atRob $\leftarrow q'$

In this example, the environment consists of a robot and five movable objects A, B, C, D, and E. Each action moveF (move free) and moveH (move when holding an object) consists of constraints (con), preconditions (pre), and effects (eff). The state consists of continuous variables (robot configuration (q), object poses (p^{obj}), and grasp pose (g)) and discrete variables (holding). This mixture of continuous and discrete variables makes the TAMP problem computationally challenging.

CHAPTER 3
FORMULATING CLASSICAL PLANNING INTO FACTOR GRAPH REPRESENTATION

A factor graph is a probabilistic graphical model that can represent various robotics problems (Dellaert 2021). In Chapter 3, we expand the use of factor graphs by proposing a method to represent classical planning problems in factor graphs.

SAS+ formulation provides multi-valued variables for a classical planning problem. The variables are used to represent the states in the factor graph. We propose two methods to represent the operators. The first method in Section 3.2 uses a single action variable which allows sequential plans. The second method in Section 3.3 groups the operators, allowing concurrent plans.

3.1 Classical Planning Example

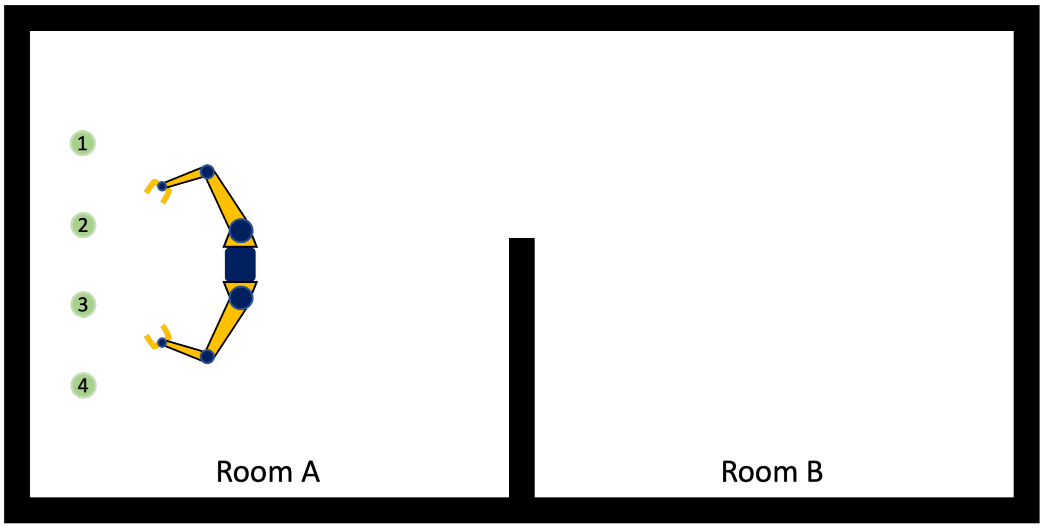


Figure 3.1: In this environment, a `Robot` can `Move` between `RoomA` and `RoomB`. Also, it has two grippers that can either `Pick` or `Drop` a `Ball`.

Gripper domain which appeared as a benchmark in International Planning Competition

1998 (IPC 1998), is visualized in Figure 3.1. The goal is to move all balls, `[ball1, ball2, ball3, ball4]`, from `RoomA` to `RoomB`. This problem is particularly interesting because it allows concurrent plans since the robot can pick and drop two balls simultaneously using its two grippers. This is similar to what we envision as a centralized multi-robot planner, a central planner controlling all physical robots.

Listing 3.1: Gripper Domain.pddl

```
(define (domain gripper-strips)
(:predicates
    (room ?r)
    (ball ?b)
    (gripper ?g)
    (at-robby ?r)
    (at ?b ?r)
    (free ?g)
    (carry ?o ?g)
)
(:action move
    :parameters (?from ?to)
    :precondition (and (room ?from) (room ?to) (at-robby ?from))
    :effect (and (at-robby ?to) (not (at-robby ?from)))
)
(:action pick
    :parameters (?obj ?room ?gripper)
    :precondition (and  (ball ?obj) (room ?room) (gripper ?gripper)
                  (at ?obj ?room) (at-robby ?room) (free ?gripper))
    :effect (and (carry ?obj ?gripper) (not (at ?obj ?room)) (not (free ?gripper)))
)
(:action drop
    :parameters  (?obj  ?room ?gripper)
    :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
                    (carry ?obj ?gripper) (at-robby ?room))
    :effect (and (at ?obj ?room) (free ?gripper) (not (carry ?obj ?gripper))))
)
```

Listing 3.2: Gripper Problem.pddl

```
(define (problem strips-gripper4)
(:domain gripper-strips)
(:objects rooma roomb ball1 ball2 ball3 ball4 left right)
(:init
    (room rooma)
    (room roomb)
    (ball ball1)
    (ball ball2)
    (ball ball3)
    (ball ball4)
    (gripper left)
    (gripper right)
    (at-robby rooma)
    (free left)
    (free right)
    (at ball1 rooma)
    (at ball2 rooma)
    (at ball3 rooma)
    (at ball4 rooma))
(:goal
    (and (at ball1 roomb) (at ball2 roomb) (at ball3 roomb)))
)
```

Listing 3.3: SAS+ Variable

```
// Sample variables section
7
begin_variable
var0
-1
5
Atom carry(ball1, right)
Atom carry(ball2, right)
Atom carry(ball3, right)
Atom free(right)
Atom carry(ball4, right)
end_variable
begin_variable
var1
-1
5
Atom carry(ball3, left)
Atom free(left)
Atom carry(ball2, left)
Atom carry(ball1, left)
Atom carry(ball4, left)
end_variable
begin_variable
var2
-1
3
Atom at(ball4, rooma)
Atom at(ball4, roomb)
<none of those>
end_variable
begin_variable
var3
-1
3
Atom at(ball3, rooma)
Atom at(ball3, roomb)
<none of those>
end_variable
begin_variable
var4
-1
3
Atom at(ball1, rooma)
Atom at(ball1, roomb)
<none of those>
end_variable
begin_variable
var5
-1
3
Atom at(ball2, rooma)
Atom at(ball2, roomb)
<none of those>
end_variable
begin_variable
var6
-1
2
Atom at-robby(roomb)
Atom at-robby(rooma)
end_variable
```

Listing 3.4: SAS+ Mutex

```
// Sample mutex section
7
begin_mutex_group
4
1 4
0 4
2 0
2 1
end_mutex_group
begin_mutex_group
4
1 0
0 2
3 0
3 1
end_mutex_group
begin_mutex_group
4
1 3
0 0
4 0
4 1
end_mutex_group
begin_mutex_group
5
1 1
1 4
1 0
1 2
1 3
end_mutex_group
begin_mutex_group
5
0 3
0 4
0 2
0 1
0 0
end_mutex_group
begin_mutex_group
2
6 1
6 0
end_mutex_group
begin_mutex_group
4
1 2
0 1
5 0
5 1
end_mutex_group
```

Listing 3.5: SAS+ Op.

```
// Sample operator section
34
begin_operator
move rooma roomb
0
1
0 6 1 0
0
end_operator
begin_operator
pick ball4 rooma left
1
6 1
2
0 1 1 4
0 2 0 2
0
end_operator
[... 31 operators omitted]
begin_operator
pick ball1 roomb right
1
6 0
2
0 0 3 0
0 4 1 2
0
end_operator

// Sample initial state
begin_state
3
1
0
0
0
0
1
end_state

// Sample goal state
begin_goal
4
2 1
3 1
4 1
5 1
end_goal
```

3.2 SAS+ Formulation to Factor Graph for Sequential Plans

PDDL (Aeronautiques et al. 1998) is a widely used propositional language that makes Planning-as-Satisfiability encoding straightforward. However, representing propositional variables in a factor graph can cause an exponential increase in the number of discrete states, making it less effective. For example, 5 propositional variables would lead to $2^5 = 32$ discrete states. Helmert showed that it is possible to automatically translate PDDL formulation into a multivariate formulation, namely SAS+ (Helmert 2009). Therefore, to reduce the cardinality of discrete states, we use a classical planning problem written in SAS+ formulation to represent as a factor graph which can then be solved using MAP inference.

Listing 3.3, Listing 3.4, and Listing 3.5 show how a classical planning problem (Gripper domain) is written in SAS+ formulation. The formulation already contains a set of multi-valued variables that define the environment's state, making it straightforward to convert them to variables in the factor graph. $State_0$ in Figure 3.2 is the direct conversion from SAS+ formulation to factor graph variables.

Each operator in the SAS+ formulation can directly be converted to a binary variable that holds $True$ if all prevail, preconditions, and effects are satisfied and $False$ otherwise. However, from Listing 3.5, we can already see that even a simple environment has 34 binary operators, which leads to $2^{34} = 17179869184$ discrete states. Instead, we create a single new variable with a cardinality of the `number of operators` like shown in Figure 3.2 as the $Action$ variable. This variable can have a value of anywhere between 0 to `number of operators`, meaning each value stands for an operator such as `move rooma roomb`, from Listing 3.5.

Along with the variables, the four following types of factors are created to represent the parsed SAS+ formulation as a factor graph:

- **Initial Factor**:

Returns true if a set of variables has predefined values from the initial state. For example, an initial factor from Figure 3.2 returns true only if `[var0=3, var1=1, var2=0, var3=0, var4=0, var5=0, var6=1]`, which is defined in Listing 3.5.

- **Mutex Factor**:

  Similar to a mutually exclusive constraint, it returns true if a set of variables have different values. A mutex factor (green, top left) connected to `var1, var2, var3` from Figure 3.2 will return true if $\texttt{var1} \neq \texttt{var2} \neq \texttt{var3}$.

- **Operator Factor**:

  Returns true if a set of variables have predefined values (prevail, precondition, effect) from the operator section. For example, the operator factor in Figure 3.2 will return true when $Action = 34$ (`move rooma roomb`) if `var0=0` (precondition) and `var7=1` (effect). Note that `var7` represents the same variable as `var0` from Listing 3.3 but in the next time step. This factor guarantees that if an action is chosen, the state of the environment is changed from the value of the precondition to the value of the effect while the value of prevail remains unchanged.

- **Frame Factor**:

  Returns true if variables not involved in the operator factor have the same values from $state_t$ to $state_{t+1}$. For example, the frame factor in Figure 3.2 will return true when $Action = 34$ (`move rooma roomb`), if [`var1=var8`, `var2=var9`, ..., `var6=var13`]. Note that variables on the right (`var8`, ..., `var13`) represent the same variables as the ones on the left (`var1`, ..., `var6`) from Listing 3.3 but in the next time step. This factor guarantees that no other variables have changed except for the variables affected by the chosen action.

- **Goal Factor**:

  Returns true if a set of variables have the predefined values from the goal state. For example, a goal factor from Figure 3.2 returns true only if `[var2=1, var3=1, var4=1, var5=1]` as defined in Listing 3.5.

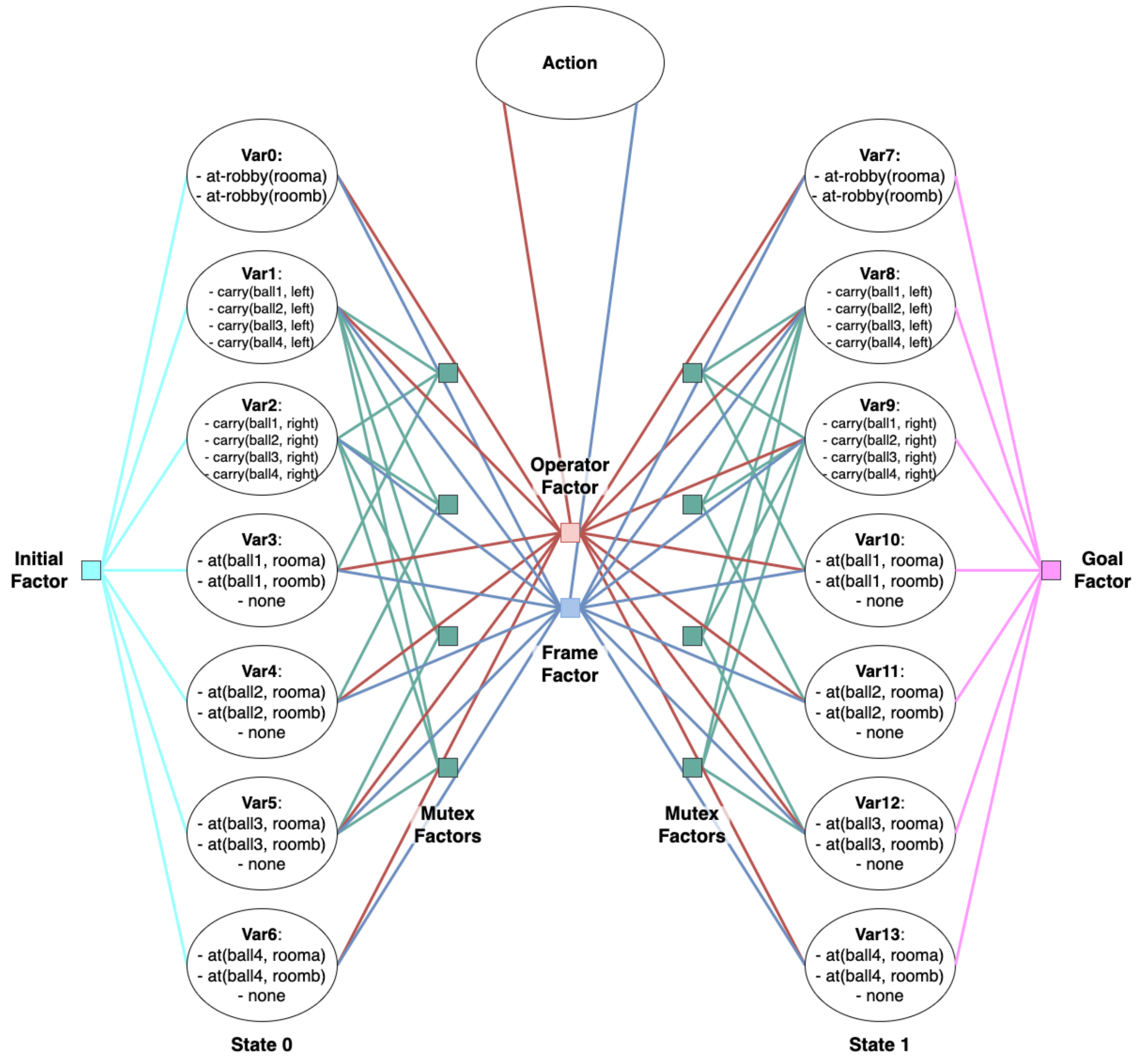


Figure 3.2: Factor graph representation of gripper planning problem with a horizon of 1 formulated for a sequential plan. Cyan and pink boxes indicate initial and goal factors, green boxes indicate mutex factors and red and blue boxes in the middle indicate operator and frame factors explained in Section 3.2. Each variable selects one value among possible values. For example `Var0` can choose `at-robby(rooma)` or `at-robby(roomb)`.

A valid plan is a combination of values assigned to variables that satisfies all factors in the factor graph. The factor graph in Figure 3.2 is created for a single horizon plan, but all factors can be copied to extend the factor graph for a longer horizon.

3.3 SAS+ Formulation to Factor Graph for Concurrent Plans

The proposed method is Section 3.2 can compute a valid sequential plan. However, concurrent planning is not supported because it uses a single action variable per timestep to avoid the exponential number of discrete states caused by a set of binary operator variables. Also, the operator factor and frame factor are connected to all variables in the timestep, meaning the factors have huge cardinality and slow the MAP inference.

To solve the aforementioned issues, we decompose the $Action$ variable in Figure 3.2 into multiple multi-valued variables as visualized in Figure 3.3. We do this by grouping the operators that involve same variables. For example, the operators in a group, **`Act0`** from Figure 3.3, [`drop ball1 rooma left`, `drop ball1 roomb left`, `pick ball1 rooma left`, `pick ball1 roomb left`, `no-op`], all involve `var0`, `var1`, `var3` from Listing 3.3.

Naturally, the groupings expose the operators that cannot be executed in parallel. In other words, at most, one operator can be chosen from a group at a single timestep. For cases where no operator is chosen, we define `no-op`, which stands for no-operation. On the other hand, operators from different groups have no mutual exclusive constraint, allowing parallel execution.

The operator factor and frame factor of each timestep in Figure 3.2 are also decomposed into multiple state factors in Figure 3.3. Each state factor is connected to a state variable in $state_t$ and $state_{t+1}$ along with grouped operator variables that change the value of the state variable through precondition and effect or maintain its value through prevail.

The advantage of decomposed factors is the reduced cardinality compared to the previous method in Section 3.2. Smaller cardinality reduces the computation needed to multiply the factors during MAP inference in the factor graph.

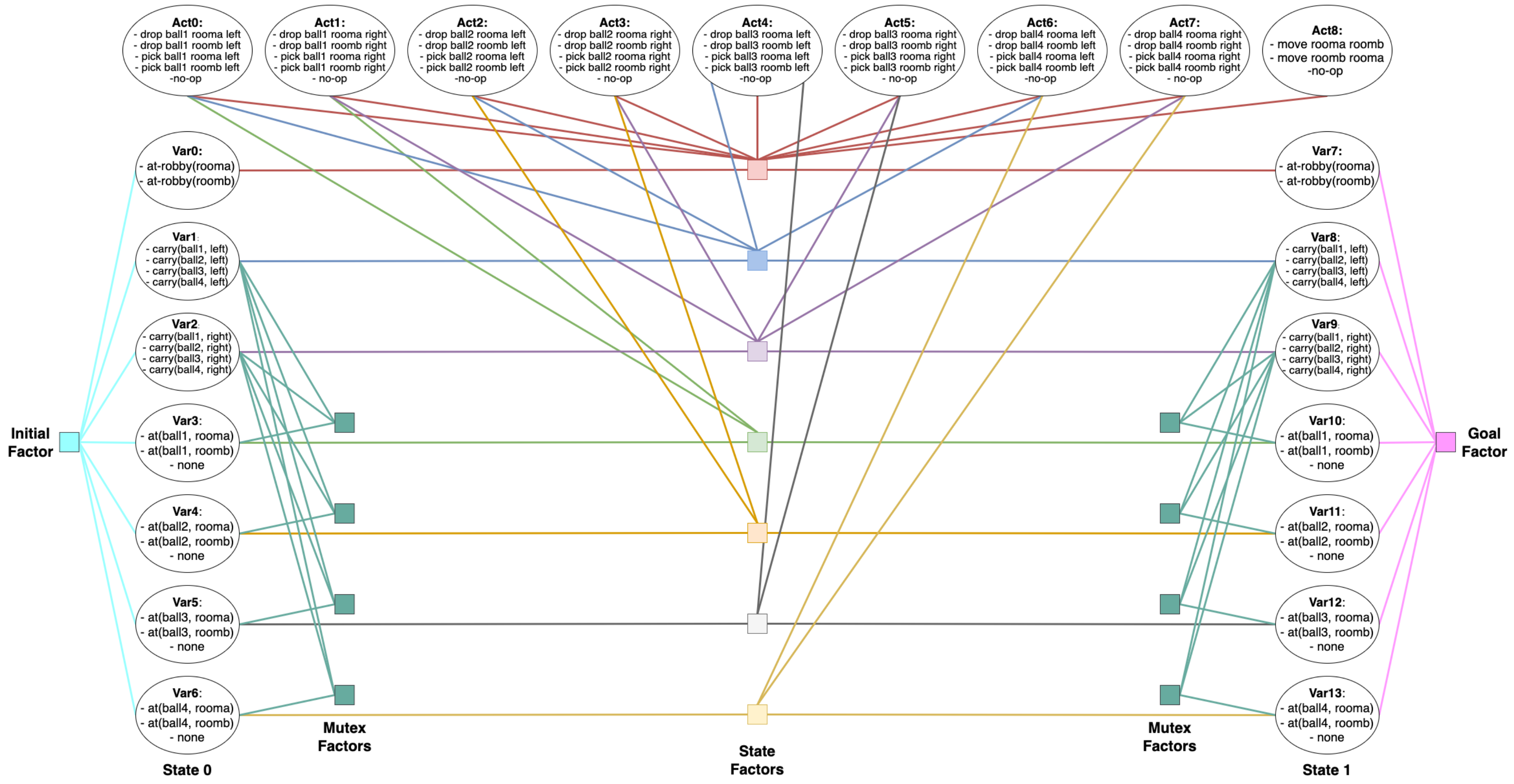


Figure 3.3: Factor graph representation of gripper planning problem with a horizon of 1 formulated to handle concurrent plans. Cyan and pink boxes indicate initial and goal factors, green boxes indicate mutex factors and boxes in the middle indicate state factors explained in Section 3.3. Each variable selects one value among possible values. For example `Var0` can choose `at-robby(rooma)` or `at-robby(roomb)`.

Although the decomposed $Action$ variable allows concurrent plans, it also drastically increase the number of possible combination of operators that can be taken in each timestep. For example, from Figure 3.3, possible combination of operators from $State_0$ to $State_1$ is $|Act0| \times |Act1| \times \cdots \times |Act8| = 5^8 \times 3$. Thus, considering all possible combinations during inference inevitably takes long. However, most combinations of operators are "invalid", that is, does not satisfy all the predefined preconditions and effects of the operators. Hence, we developed a discrete probabilistic factor optimized for sparsity explained in Chapter 4, which ignores the "invalid" states during MAP inference.

CHAPTER 4
DISCRETE PROBABILISTIC FACTOR OPTIMIZED FOR SPARSITY

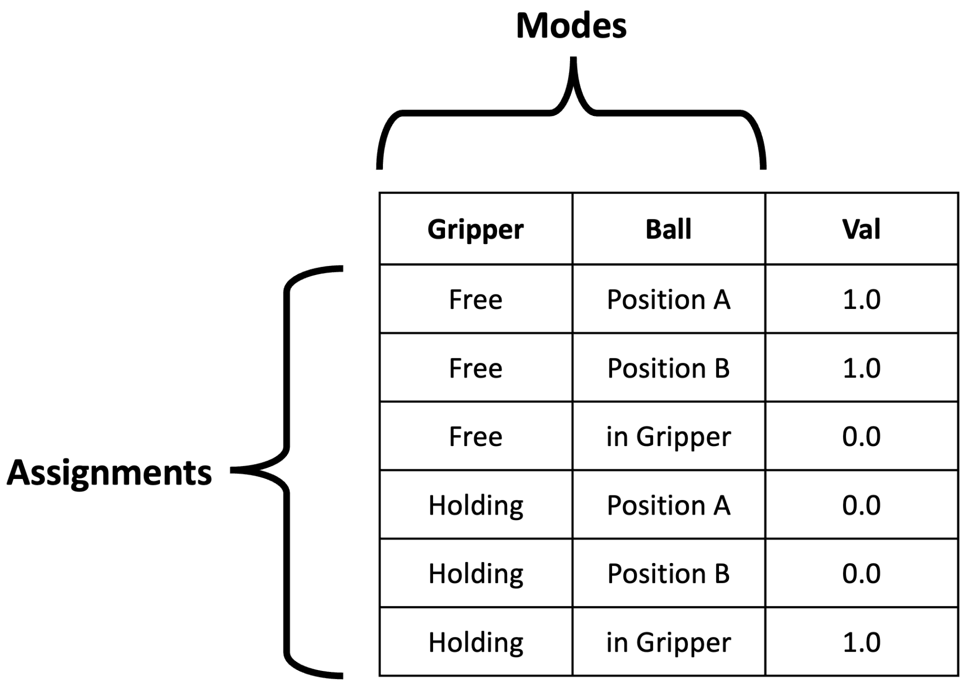
Maximum a posteriori (MAP) inference algorithm maximizes the product of all factors in the factor graph. This leads to intractable computation because of the exponential increase in the number of modes with respect to the number of assignments. Yet, when classical planning problems are represented in factor graphs, most assignments are invalid because at least one of three constraints (preconditions, effects, or mutex constraints) is violated. Figure 4.2 illustrates why assignments in a simple classical planning problem are invalid due to a mutex constraint. Assignment 4 in Figure 4.2 is an example of an invalid state. If a `Ball` is at `Position A`, the `Gripper` should not be `Holding` anything making the state invalid and thus have a value of 0.0.

Definition 1 *The variables of the factor graph from Chapter 3 is defined as* ***`Mode`****, and the values assigned to the variables are defined as* ***`Assignment`****, as shown in Figure 4.1.*

Anything multiplied by 0 is 0; therefore, multiplication between factors should be able to ignore invalid assignments. To this end, we developed a discrete probabilistic factor optimized for sparsity which we call **TableFactor**.

4.1 Data Structure

The underlying data structure of TableFactor is a sparse vector. Sparse vector uses a compressed column storage scheme (Gustavson 1972) where only the nonzero elements are stored along with the indices of nonzero elements. This storage scheme is widely used because it does not have any assumptions about the structure of the sparsity. Specifically, open source implementation (Guennebaud et al. 2010) of sparse vector, which uses compressed column storage. A comparison of a vector and a sparse vector is shown in Fig-


Modes
Assignments

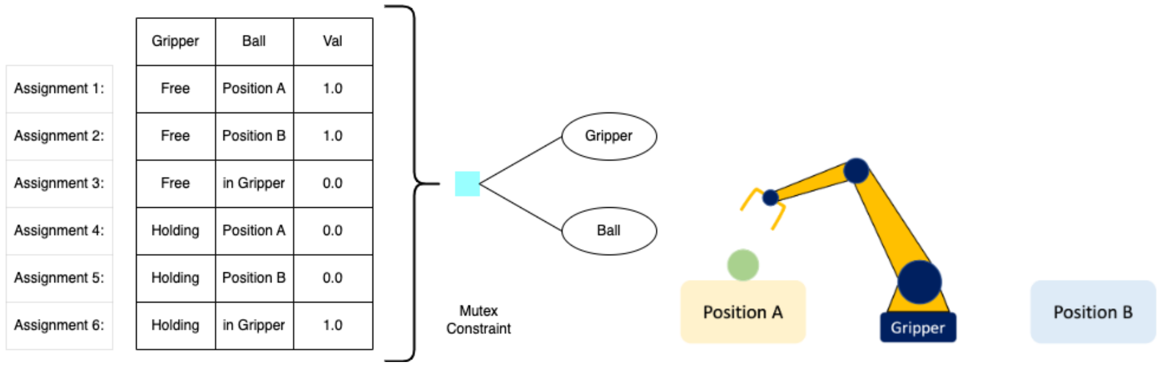
Gripper	Ball	Val
Free	Position A	1.0
Free	Position B	1.0
Free	in Gripper	0.0
Holding	Position A	0.0
Holding	Position B	0.0
Holding	in Gripper	1.0

Figure 4.1: Discrete states are referred to as modes and discrete values are referred to as assignments.

ure 4.3. The storing order of the values is a Cartesian product of the modes. For example, if `Gripper` has a cardinality of 2 and `Ball` has a cardinality of 3, the Cartesian product of the two modes $\texttt{Gripper} \times \texttt{Ball} = \{(0,0)(0,1),(0,2),(1,0),(1,1),(1,2)\}$. The values are stored in the same order leading to the result in Figure 4.4. By utilizing the ordering of the values, TableFactor does not have to save all the assignments of the modes, making it more compact. Nevertheless, assignments of the modes have to be found at some point during the computation. Finding an assignment of a mode can be done in O(1) using the Lazy Cartesian Product algorithm in Equation 4.1.

4.2 Sparse Factor Multiplication

At the core of MAP estimation in a factor graph is the multiplication of factors. This means the MAP algorithm will also become faster if multiplication becomes more efficient. However, sparse factor multiplication is challenging because the number of non-zero elements and the non-zero pattern of the output factor is unknown before computation. The most



	Gripper	Ball	Val
Assignment 1:	Free	Position A	1.0
Assignment 2:	Free	Position B	1.0
Assignment 3:	Free	in Gripper	0.0
Assignment 4:	Holding	Position A	0.0
Assignment 5:	Holding	Position B	0.0
Assignment 6:	Holding	in Gripper	1.0

Figure 4.2: In this environment, a `Gripper` can either `Hold` a ball or be `free`. The `Ball` can be at `Position A`, `Position B`, or `in the gripper`. This figure shows why a factor in a classical planning problem is sparse, that is, contain invalid assignments.

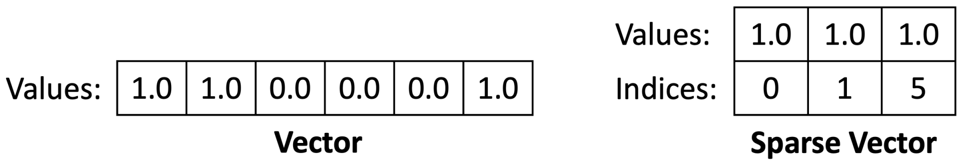


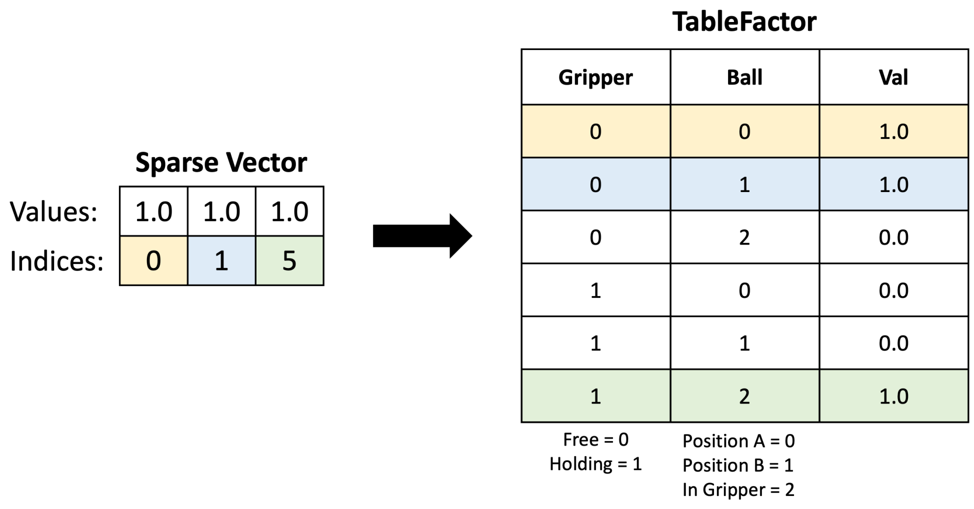
Figure 4.3: The figure shows how a vector stores its elements and how a sparse vector stores its elements.

efficient way would be to look at only the non-zero elements of both factors during computation. The workflow of the multiplication process between two TableFactors is visualized in Figure 4.5.

Multiplication of two factors is finding all combinations of assignments that both `Factor1` and `Factor2` will have non-zero elements on. For example, in Figure 4.5 the assignment {`Var1:0, Var2:3, Var3:0, Var4:1, Var5:0, Var6:0`} will output 3.0 in `Factor1` and 4.0 in `Factor2`.To find such an assignment, the multiplication process is divided into three stages: 1. Input Processing, 2. Converting to Hash Table, 3. Find Index and Insert.

4.2.1 Input Processing

In the input processing stage, we identify the contract and free modes of the two factors being multiplied. **`Contract modes`** are the modes that the two factors have in common. For example, in Figure 4.5 `Factor1` and `Factor2` both have `Var5` and `Var6` in, making



Values:	1.0	1.0	1.0
Indices:	0	1	5

Gripper	Ball	Val
0	0	1.0
0	1	1.0
0	2	0.0
1	0	0.0
1	1	0.0
1	2	1.0

Figure 4.4: Sparse Vector assumes Cartesian product storage order of the TableFactor. The figure shows why TableFactor does not have to save all the assignments of the modes. For example, `Val`=1.0 at `Gripper=0, Ball=0` in TableFactor is equivalent of `Val`=1.0 at index 0 of the Sparse Vector.

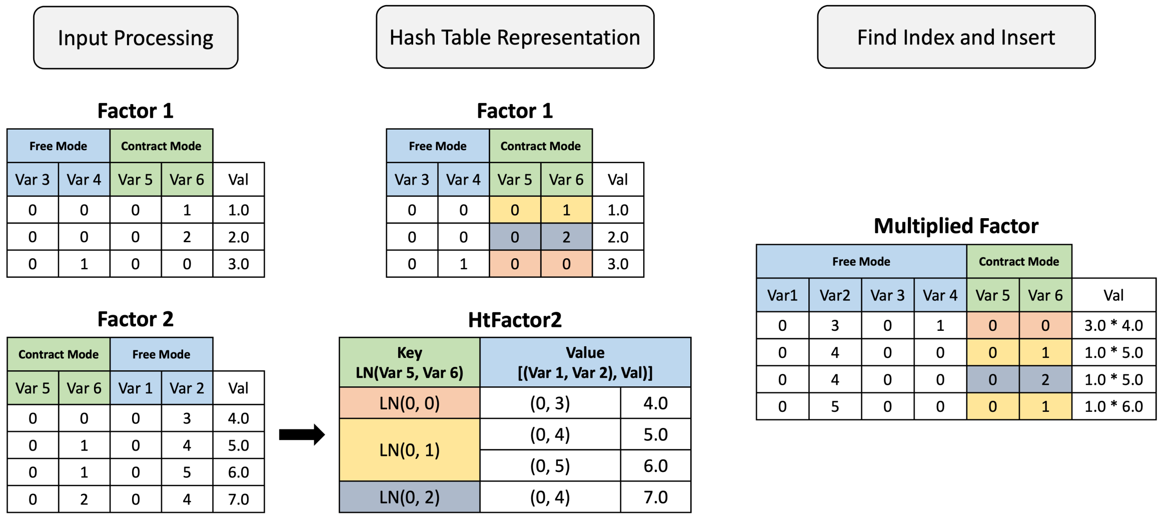
them `contract modes`.

`Free modes` are the modes that are exclusive to one factor. In Figure 4.5 `Factor1` has `Var3` and `Var4` but `Factor2` doesn't, and `Factor2` has `Var1` and `Var2` but `Factor1` doesn't. This makes the four modes, `Var1, Var2, Var3, Var4` all `free modes`.

4.2.2 Converting to Hash Table

A naive approach to multiplying two factors would be to look at all non-zero elements of the multiplier factor for each non-zero element of the multiplicand factor, making the time complexity roughly $O(nnz_{f1} \times nnz_{f2})$. This approach involves looking at non-zero elements that don't have the matching assignments of contract modes leading to unnecessary computing, which is visualized in Figure 4.6.

To avoid this, we convert the multiplier factor, `Factor2`, into a hash table using the assignments of contract modes as a key and a list of assignments of free modes and non-zero elements as values explained in Figure 4.5. The hash table `HtFactor2` allows us



Factor 1

Free Mode		Contract Mode		
Var 3	Var 4	Var 5	Var 6	Val
0	0	0	1	1.0
0	0	0	2	2.0
0	1	0	0	3.0

Factor 2

Contract Mode		Free Mode		
Var 5	Var 6	Var 1	Var 2	Val
0	0	0	3	4.0
0	1	0	4	5.0
0	1	0	5	6.0
0	2	0	4	7.0

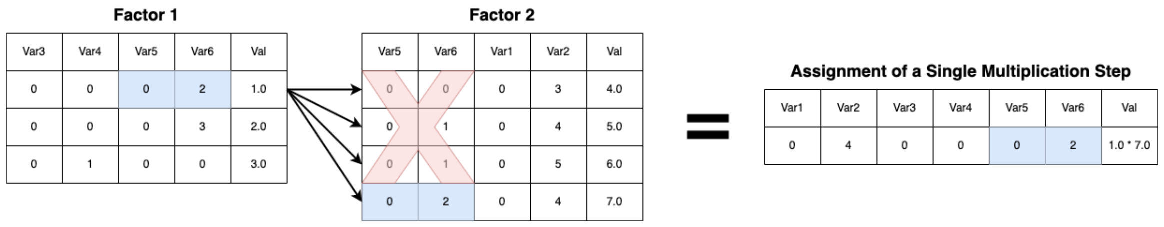
HtFactor2

Key LN(Var 5, Var 6)	Value [(Var 1, Var 2), Val)]	
LN(0, 0)	(0, 3)	4.0
LN(0, 1)	(0, 4)	5.0
	(0, 5)	6.0
LN(0, 2)	(0, 4)	7.0

Multiplied Factor

Free Mode				Contract Mode		
Var1	Var2	Var 3	Var 4	Var 5	Var 6	Val
0	3	0	1	0	0	3.0 * 4.0
0	4	0	0	0	1	1.0 * 5.0
0	4	0	0	0	2	1.0 * 5.0
0	5	0	0	0	1	1.0 * 6.0

Figure 4.5: Workflow of the multiplication process between two TableFactors. We first identify contract modes and free modes. Next, we convert `Factor2` into hash table representation `HtFactor2`. Finally we calculate the index of assignments with agreeing contract modes and insert multiplied values to the `Multiplied Factor`



Factor 1

Var3	Var4	Var5	Var6	Val
0	0	0	2	1.0
0	0	0	3	2.0
0	1	0	0	3.0

Factor 2

Var5	Var6	Var1	Var2	Val
0	0	0	3	4.0
0	1	0	4	5.0
0	1	0	5	6.0
0	2	0	4	7.0

Assignment of a Single Multiplication Step

Var1	Var2	Var3	Var4	Var5	Var6	Val
0	4	0	0	0	2	1.0 * 7.0

Figure 4.6: Workflow of the naive multiplication process between two TableFactors leads to inefficiency.

to look at values with matching assignments of contract modes reducing the unnecessary compute and making the multiplication time complexity $O(nnz_{f1} + nnz_{f2})$ at the cost of additional compute from the hash table conversion.

To convert `Factor2` to `HtFactor2`, we first loop over the non-zero elements of `Factor2` (Line 3). For each non-zero element, we find the assignments of contract modes and free modes using Lazy Cartesian Product (Equation 4.1) and calculate the large-number representation of the assignment (Line 4, Line 5, Line 6). Using the large-number representation as a key, the assignments of free modes and the non-zero element are inserted into a vector (Line 8, Line 10). This results in groupings of assignments of free modes and non-zero elements like in Figure 4.5 `HtFactor2` has $LN(0, 1)$ as a key and a vector $[((0,4),5.0),((0,5),6.0)]$ as a value. Algorithm 2 presents a pseudo-code of how the conversion can be done in $O(nnz_{f2} \times M_{f2})$ where nnz stands for the number of non-zero elements, and M stands for the number of modes.

We use large-number representation because it is time-consuming to match the assignments of contract modes represented as tuples. The algorithm for converting assignments to large-number representation is presented in Algorithm 1. We loop over the $assignments$ in reverse order (Line 4). For each mode in the $assignments$, the cardinality of the mode is multiplied to $card$ (Line 5) and $card$ is added to ln (Line 6). Time complexity of Algorithm 1 is $O(M)$ where M stands for the number of modes in the $assignments$. The final return value ln is a unique and decodable "large" number, that is, each assignment has its own unique large number representation and it is possible to decode ln to obtain $assignments$ if we know the cardinality of the modes.

Algorithm 1 Large-Number Representation

```
1: procedure LN($assignments$)
2:     $ln = 0$
3:     $card = 1$
4:     for $k \in$ reversed($assignments$) do
5:         $card$ $*=$ cardinality of $k$
6:         $ln$ $+=$ $card$
7:     end for
8: end procedure
9: return $ln$
```

4.2.3 Find Index and Insert

After converting `Factor2` to `HtFactor2`, all combinations of assignments with agreeing contract modes are inserted into a new `Multiplied Factor` along with the multiplied non-zero elements. After the conversion of `Factor2` (Line 6) we loop over the non-zero elements of `Factor1` (Line 7). For each non-zero element, the large-number representation of the contract modes' assignments is calculated (Line 8, Line 9). If the large number exists as a key in the converted hash table, we loop over the values which contain

Algorithm 2 Hash Table Conversion

```
1:  procedure HashTableConversion(contract modes, free modes)
2:      $map \leftarrow$ Empty Hash Table
3:      for $nnz \in factor$ do
4:          $contractAssignment =$ LazyCartesianProduct(contract modes)
5:          $contractLN =$ LN($contractAssignment$)
6:          $freeAssignment =$ LazyCartesianProduct(free modes)
7:          if $contractLN$ not in $map$ then
8:              $map[contractLN] = [(freeAssignment, nnz)]$
9:          else
10:             $map[contractLN].add((freeAssignment, nnz))$
11:         end if
12:     end for
13:     return $map$
14: end procedure
```

`Factor2`'s free modes' assignments and non-zero elements (Line 13, Line 14). Next, we combine all assignments from contract modes, `Factor1`'s free modes, and `Factor2`'s free modes to create a union of assignments (Line 15) and find the corresponding index using the Inverse of Lazy Cartesian Product Equation 4.2 (Line 16). Finally, multiplied non-zero element is inserted into the calculated index (Line 17). The pseudo-code of the overall multiplication process is presented in Algorithm 3. The time complexity of Algorithm 3 is $O(nnz_{f1} \times M_{union} \times V + nnz_{f2} \times M_{f2})$. nnz_{f1} and nnz_{f2} are number of nonzero elements in `Factor1` and `Factor2`. M_{union} and M_{f2} are number of modes in union of `Factor1` and `Factor2` and number of modes in `Factor2`. Finally, V is the number of elements in $map[contractLN]$.

4.3 Lazy Cartesian Product

The Lazy Cartesian Product algorithm was used throughout the multiplication process to obtain a mode's assignment. Formally, it finds the n^{th} element of the Cartesian product of

Algorithm 3 Find Index and Insert

```
1: procedure MULTIPLY($factor1$, $factor2$)
2:     $contract \leftarrow$ Contract modes between Factor1 and Factor2
3:     $freeF1 \leftarrow$ Free modes of Factor1
4:     $freeF2 \leftarrow$ Free modes of Factor2
5:     $sparseVector \leftarrow$ Empty Sparse Vector
6:     $mapF2 = factor2$.HashTableConversion($contract$, $freeF2$)
7:     for $nnzF1 \in factor1$ do
8:         $contractAssign =$ LazyCartesianProduct($contract$)
9:         $contractLN =$ LN($contractAssign$)
10:        if $contractLN$ not in $map$ then
11:            continue
12:        else
13:            $freeAssignF1 =$ LazyCartesianProduct($freeF1$)
14:            for $(freeAssignF2$, $nnzF2) \in map[contractLN]$ do
15:                $unionAssign = [contractAssign$, $freeAssignF1$, $freeAssignF2]$
16:                $index =$ InverseLazyCartesianProduct($unionAssign$)
17:                $sparseVector[index] = nnzF1 \times nnzF2$
18:            end for
19:        end if
20:    end for
21:    return TableFactor($unionKeys$, $sparseVector$)
22: end procedure
```

arrays $Z = A \times B \times \cdots \times N$.

$$Z(n) = \begin{bmatrix} A\left[\left\lfloor \frac{n}{|B||C|\cdots|N|} \right\rfloor \bmod |A|\right], \\ B\left[\left\lfloor \frac{n}{|C||D|\cdots|N|} \right\rfloor \bmod |B|\right], \\ \vdots \\ N\left[\left\lfloor \frac{n}{1} \right\rfloor \bmod |N|\right] \end{bmatrix} \tag{4.1}$$

Since dividends and cardinalities of the modes are pre-computed and stored in a map when the TableFactor is first created, finding an assignment of a mode can be done in $O(1)$.

4.4 Inverse Lazy Cartesian Product

The Inverse Lazy Cartesian Product algorithm is used in the Algorithm 3 to find the index of the assignments with agreeing contract modes. Formally, it finds the index n given the assignments of the modes.

$$n = \left\lfloor \frac{n}{|B||C|\cdots|N|} \right\rfloor \times |A| + \left\lfloor \frac{n}{|C||D|\cdots|N|} \right\rfloor \times |B| + \cdot + \left\lfloor \frac{n}{1} \right\rfloor \times |N| \tag{4.2}$$

4.5 Multiplication Benchmark

We compare the execution time of the multiplication between two TableFactors versus two DecisionTreeFactors. DecisionTreeFactor is a discrete probabilistic factor used in GT-SAM (Dellaert et al. 2017b), which uses factor graphs and Bayes networks as the underlying computing paradigm for various optimization problems. The experiments were run on Apple M1 Max processor with 10 cores and 32 GB of memory.

We measure the execution time of the multiplication with various numbers of elements and the density. Figure 4.7 shows that regardless of the size of the factors, the execution time is significantly reduced when the factors are sparse, meaning when 90% of elements are 0's. The execution time of multiplication between DecisionTreeFactors is shown in

Figure 4.8. Although execution time reduces when the factors are sparse, the difference between sparse and dense execution time is not as large as TableFactors.

Finally, we directly compare the execution time of multiplication for both types of factors in Figure 4.9. The execution time is measured ten times and averaged. If the factors are dense with all non-zero elements, the TableFactor performs slightly better. As the factors get sparser, we the execution time of TableFactor decreases leading to massive speedup at 90% sparsity where TableFactor is around 19.6 times faster. It is important to note that the factors used for classical planning can oftentimes be even sparser than 90%.

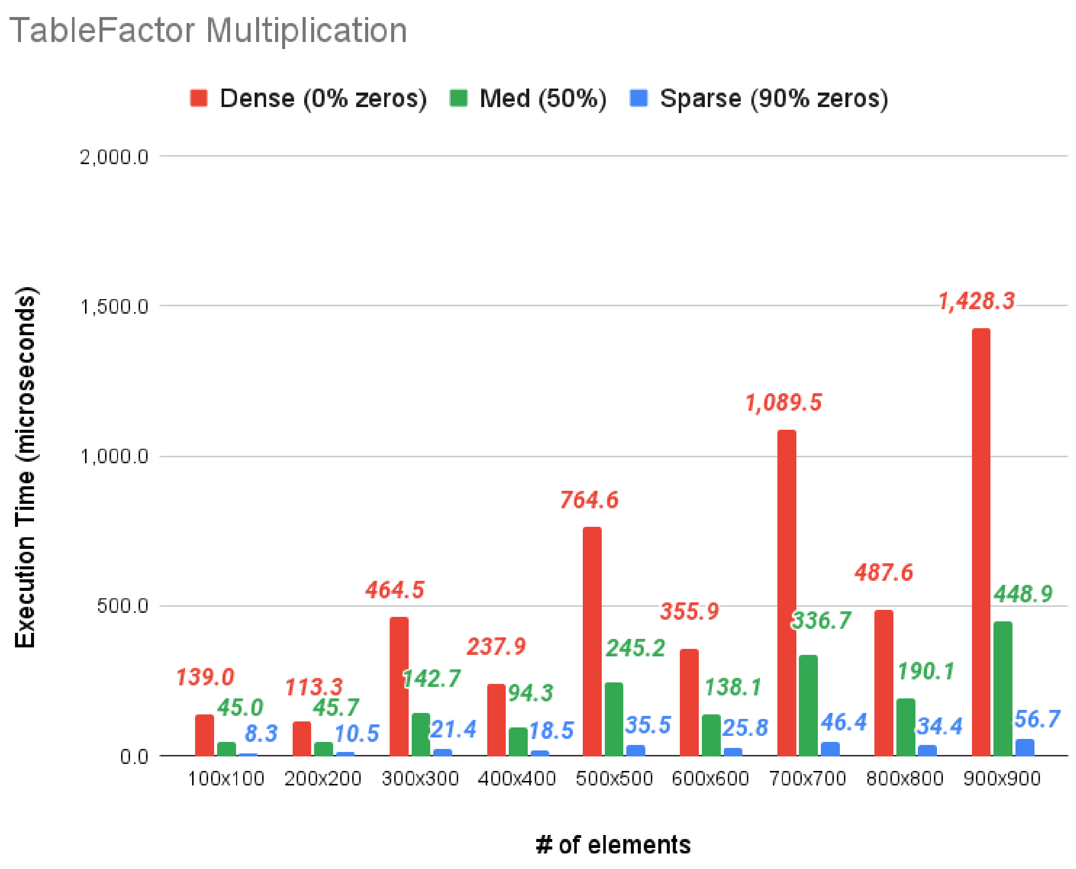


Figure 4.7: Execution time of the multiplying two TableFactors measured in microseconds. y-axis is the measured time, and x-axis is the number of maximum non-zero elements in the TableFactor. Dense means all elements are non-zeros, Medium means 50% of elements are non-zeros, and Sparse means 90% of elements are non-zero.

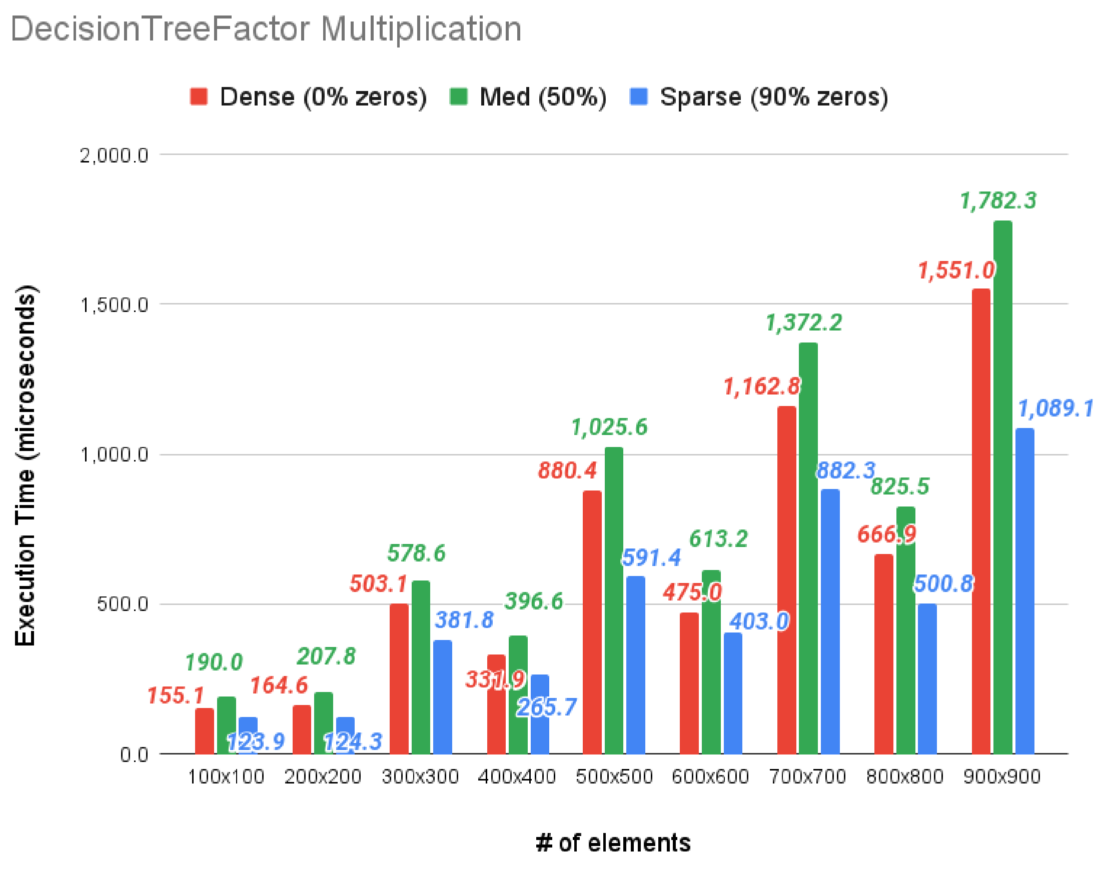


Figure 4.8: Execution time of the multiplying two DecisionTreeFactors measured in microseconds. y-axis is the measured time, and x-axis is the number of maximum non-zero elements in the TableFactor. Dense means all elements are non-zeros, Medium means 50% of elements are non-zeros, and Sparse means 90% of elements are non-zero.

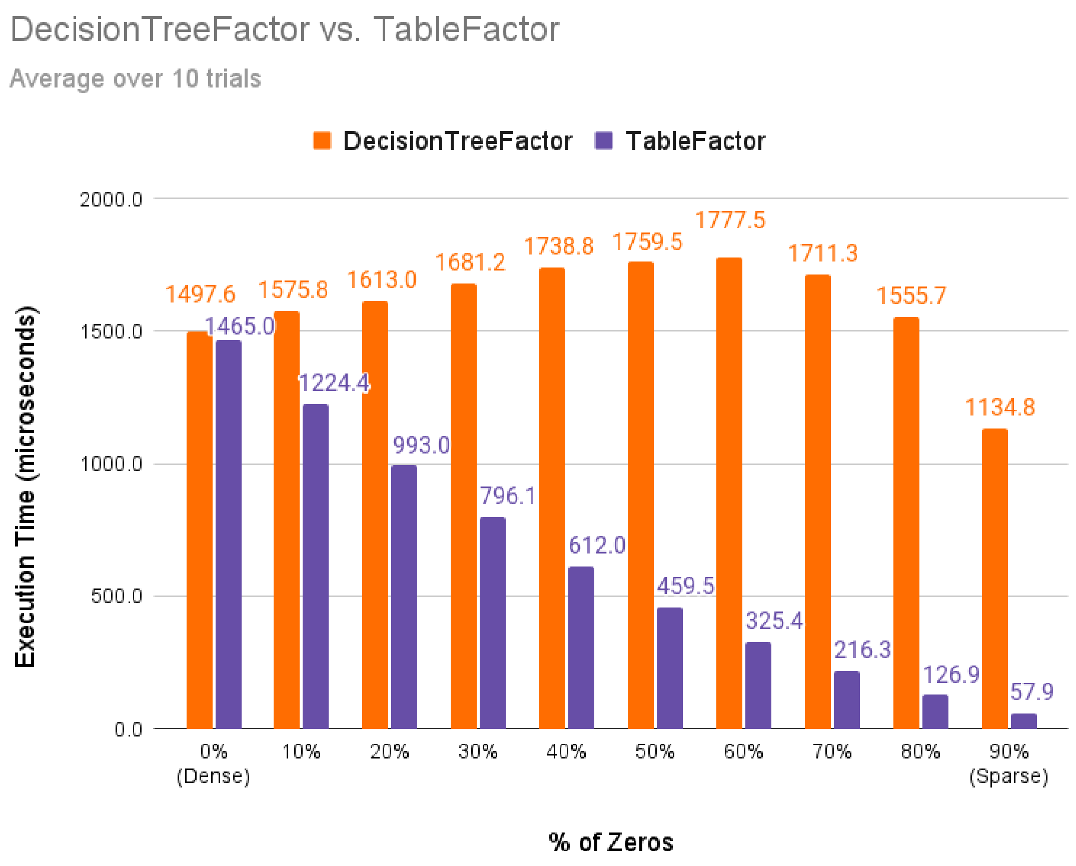


Figure 4.9: Execution time comparison between TableFactor and a discrete probabilistic factor used in GTSAM called DecisionTreeFactor. y-axis is measured time in microseconds, and x-axis is the sparsity of the factor from dense (left) to sparse (right).

CHAPTER 5
CLASSICAL PLANNING BENCHMARKS

In Chapter 5, we test the proposed method of representing classical planning problems in factor graphs from Section 3.3 on three domains: Gripper in Section 5.1, Multi-Robot in Section 5.2, and Delivery Robot in Section 5.3. In all experiments, we used off-the-shelf factor graph-based solver GTSAM (Dellaert et al. 2017b) with TableFactor integrated.

5.1 Concurrent Plan in Gripper Domain

Gripper domain first appeared as a classical planning problem benchmark in IPC 1998. It is widely used as a benchmark to measure how well the planner performs in terms of correctness and efficiency (Seipp et al. 2016; Francès Medina et al. 2019; Patrizi et al. 2011). Also, concurrent plans on gripper domain can be found which can shorten the plan horizon. Therefore, we test our proposed formulation on a factor graph which uses TableFactor on this domain by measuring the time to obtain a valid plan with varying plan horizons which is reported in Figure 5.1.

We show the benefit of concurrent plans over sequential plans in Listing 5.1 and Listing 5.2. Sequential plan was obtained by using the method from Section 3.2 and the concurrent plan was obtained by using the method from Section 3.3. The planning problem addressed by both Listing 5.1 and Listing 5.2 is to move `ball1, ball2, ball3, ball4` from `rooma` to `roomb`. Concurrent plan can reach the goal state in plan length of 7 whereas sequential plan needs 13.

Listing 5.1: Sequential

```
// Valid Sequential Plan
pick ball1 rooma left
move rooma roomb
drop ball1 roomb left
move roomb rooma
pick ball2 rooma left
move rooma roomb
drop ball2 roomb left
move roomb rooma
pick ball3 rooma left
pick ball4 rooma right
move rooma roomb
drop ball3 roomb left
drop ball4 roomb right
```

Listing 5.2: Concurrent

```
// Valid Concurrent Plan
pick ball1 rooma left, pick ball2 rooma right
move rooma roomb
drop ball1 roomb left, drop ball2 roomb right
move roomb rooma
pick ball3 rooma left, pick ball4 rooma right
move rooma roomb
drop ball3 roomb left, drop ball4 roomb right
```

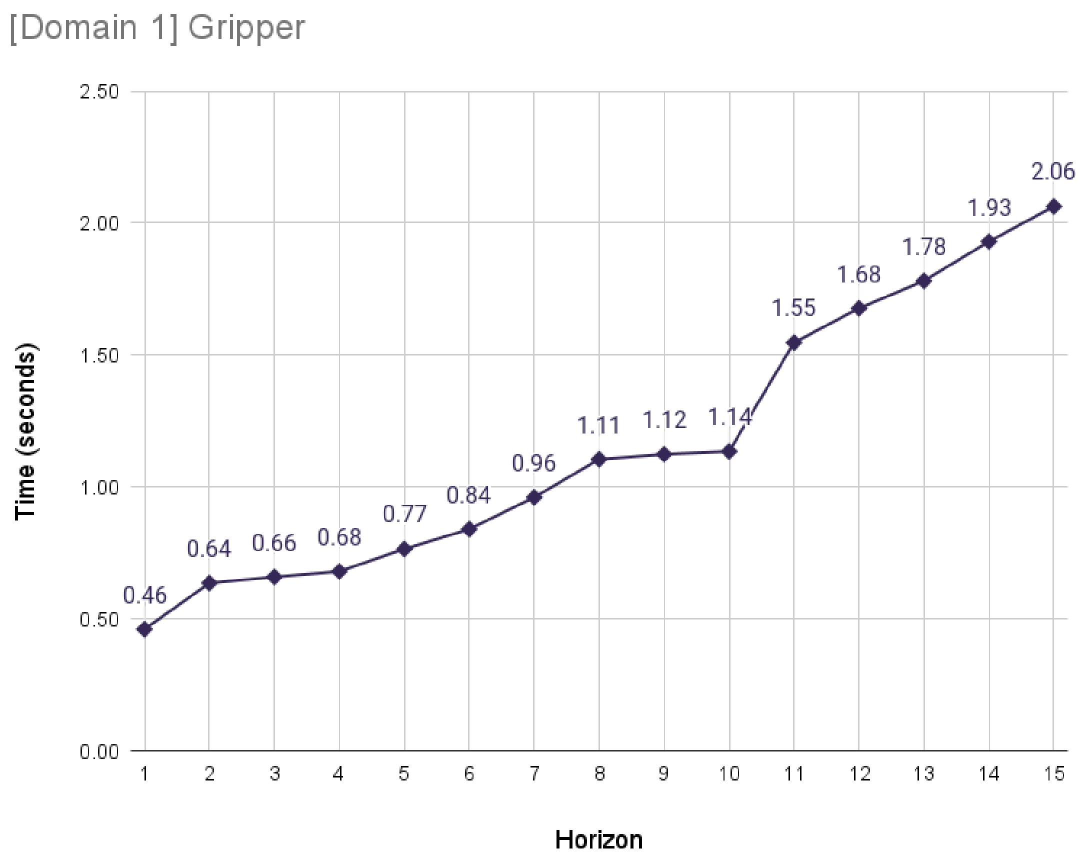


Figure 5.1: Time taken to obtain a valid plan in seconds with respect to the pre-defined plan horizon in gripper domain.

5.2 Concurrent Plan in Multi-Robot Domain

Concurrent plans are useful in centralized multi-robot environment where the central planner has control over multiple physical robots. This is because in multi-robot settings, multiple robots can execute an action in parallel at any given timestep. Hence, we created an environment to test the proposed formulation in multi-robot setting. The created environment is visualized in Figure 5.2. In this environment, a central planner can control two 3-link manipulators and a mobile base and the goal is to move the purple object around discretized positions ($p1, p2, p3, p4$). As in Section 5.1, time taken to obtain a valid plan was measured while varying the plan horizon and reported in Figure 5.3.

Listing 5.3: Domain.pddl

```
(define (domain multi-robot)
    (:requirements :adl)
    (:types arm base object position)
    (:predicates
        (arm-carrying ?a - arm ?o - object)
        (arm-is-free ?a - arm)
        (gripper-at ?a - arm ?p - position)
        (base-carrying ?b - base ?o - object)
        (base-is-free ?b - base)
        (base-at ?b - base ?p - position)
        (reachable-by-arm ?p - position ?a - arm)
        (reachable-by-base ?p - position ?b - base)
        (object-at-pos ?p - position ?o - object))

    (:action move_gripper
        :parameters (?a - arm ?p1 ?p2 - position)
        :precondition
            (and (gripper-at ?a ?p1) (reachable-by-arm ?p2 ?a))
        :effect
            (and (not (gripper-at ?a ?p1)) (gripper-at ?a ?p2)))

    (:action grab
        :parameters (?a - arm ?o - object ?p - position)
        :precondition
            (and (arm-is-free ?a) (object-at-pos ?p ?o) (gripper-at ?a ?p))
        :effect
            (and (not (arm-is-free ?a)) (not (object-at-pos ?p ?o))
                 (arm-carrying ?a ?o) (gripper-at ?a ?p)))

    (:action release
        :parameters (?a - arm ?o - object ?p - position)
        :precondition
            (and (arm-carrying ?a ?o) (gripper-at ?a ?p))
        :effect
            (and (not (arm-carrying ?a ?o)) (arm-is-free ?a)
                 (object-at-pos ?p ?o) (gripper-at ?a ?p)))

    (:action move_base
        :parameters (?b - base ?p1 ?p2 - position)
        :precondition
            (and (base-at ?b ?p1) (reachable-by-base ?p2 ?b))
        :effect
            (and (not (base-at ?b ?p1)) (base-at ?b ?p2)))

    (:action load
        :parameters (?b - base ?o - object ?p - position)
        :precondition
            (and (base-is-free ?b) (object-at-pos ?p ?o) (base-at ?b ?p))
        :effect
            (and (not (base-is-free ?b)) (not (object-at-pos ?p ?o))
                 (base-carrying ?b ?o) (base-at ?b ?p)))

    (:action unload
        :parameters (?b - base ?o - object ?p - position)
        :precondition
            (and (base-carrying ?b ?o) (base-at ?b ?p))
        :effect
            (and (not (base-carrying ?b ?o)) (base-is-free ?b)
                (object-at-pos ?p ?o) (base-at ?b ?p)))
)
```

Listing 5.4: Problem.pddl

```
(define (problem PROBLEM_X)
    (:domain multi-robot)
    (:objects
        a1 a2 - arm
        base - base
        object - object
        p1 p2 p3 p4 - position)
    (:init  (arm-is-free a1)
            (arm-is-free a2)
            (base-is-free base)
            (object-at-pos p1 object)
            (reachable-by-arm p1 a1)
            (reachable-by-arm p2 a1)
            (reachable-by-arm p3 a2)
            (reachable-by-arm p4 a2)
            (gripper-at a1 p2)
            (gripper-at a2 p4)
            (reachable-by-base p2 base)
            (reachable-by-base p3 base)
            (base-at base p3))
    (:goal (object-at-pos p4 object))
)
```

Listing 5.5: Variables

```
//Multi-Robot Variables
7
begin_variable
var0
-1
2
Atom gripper-at(a2, p3)
Atom gripper-at(a2, p4)
end_variable
begin_variable
var1
-1
2
Atom gripper-at(a1, p1)
Atom gripper-at(a1, p2)
end_variable
begin_variable
var2
-1
2
Atom base-at(base, p2)
Atom base-at(base, p3)
end_variable
begin_variable
var3
-1
2
Atom arm-is-free(a1)
NegatedAtom arm-is-free(a1)
end_variable
begin_variable
var4
-1
2
Atom arm-is-free(a2)
NegatedAtom arm-is-free(a2)
end_variable
begin_variable
var5
-1
2
Atom base-is-free(base)
NegatedAtom base-is-free(base)
end_variable
begin_variable
var6
-1
7
Atom arm-carrying(a1, object)
Atom arm-carrying(a2, object)
Atom base-carrying(base, object)
Atom object-at-pos(p1, object)
Atom object-at-pos(p2, object)
Atom object-at-pos(p3, object)
Atom object-at-pos(p4, object)
end_variable
```

Listing 5.6: Mutex.

```
//Multi-Robot Mutex
3
begin_mutex_group
2
6 0
3 0
end_mutex_group
begin_mutex_group
2
6 1
4 0
end_mutex_group
begin_mutex_group
2
6 2
5 0
end_mutex_group
// Initial and Goal states
begin_state
1
0
1
0
0
0
3
end_state
begin_goal
1
6 6
end_goal
```

Listing 5.7: Operators

```
// Multi-Robot Operators
18
begin_operator
grab a1 object p1
1
1 0
2
0 6 3 0
0 3 0 1
1
end_operator
begin_operator
grab a1 object p2
1
1 1
2
0 6 4 0
0 3 0 1
1
end_operator
begin_operator
grab a2 object p3
1
0 0
2
0 6 5 1
0 4 0 1
1
end_operator
begin_operator
grab a2 object p4
1
0 1
2
0 6 6 1
0 4 0 1
1
end_operator
[... 13 operators omitted]
begin_operator
unload base object p3
1
2 1
2
0 6 2 5
0 5 -1 0
1
end_operator
```

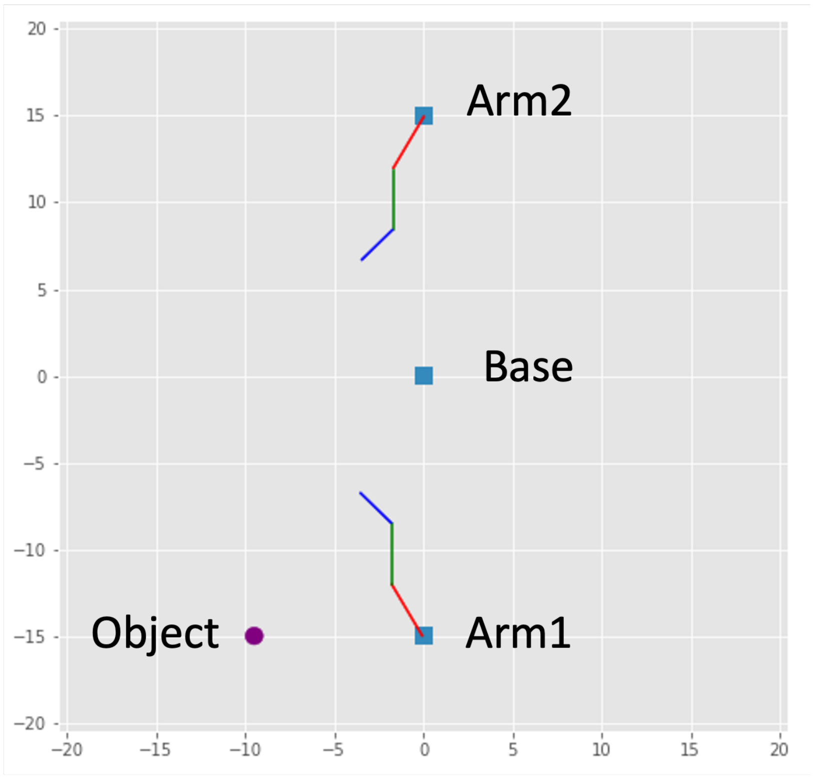


Figure 5.2: 2D multi-robot environment with two 3-link arms and one mobile base

5.3 TableFactor versus DecisionTreeFactor in Classical Planning

In our last experiment, we test the effectiveness of using TableFactors by solving for a valid plan using two types of factor graphs, one which uses TableFactor and one which uses DecisionTreeFactor. We test on delivery robot domain (Poole et al. 2010) where the goal is generate a valid plan for the robot to deliver coffee or mail. Sine the robot has two grippers, this domain also allows concurrent plans to be found. The comparison of execution time to obtain valid plan is visualized in Figure 5.4.

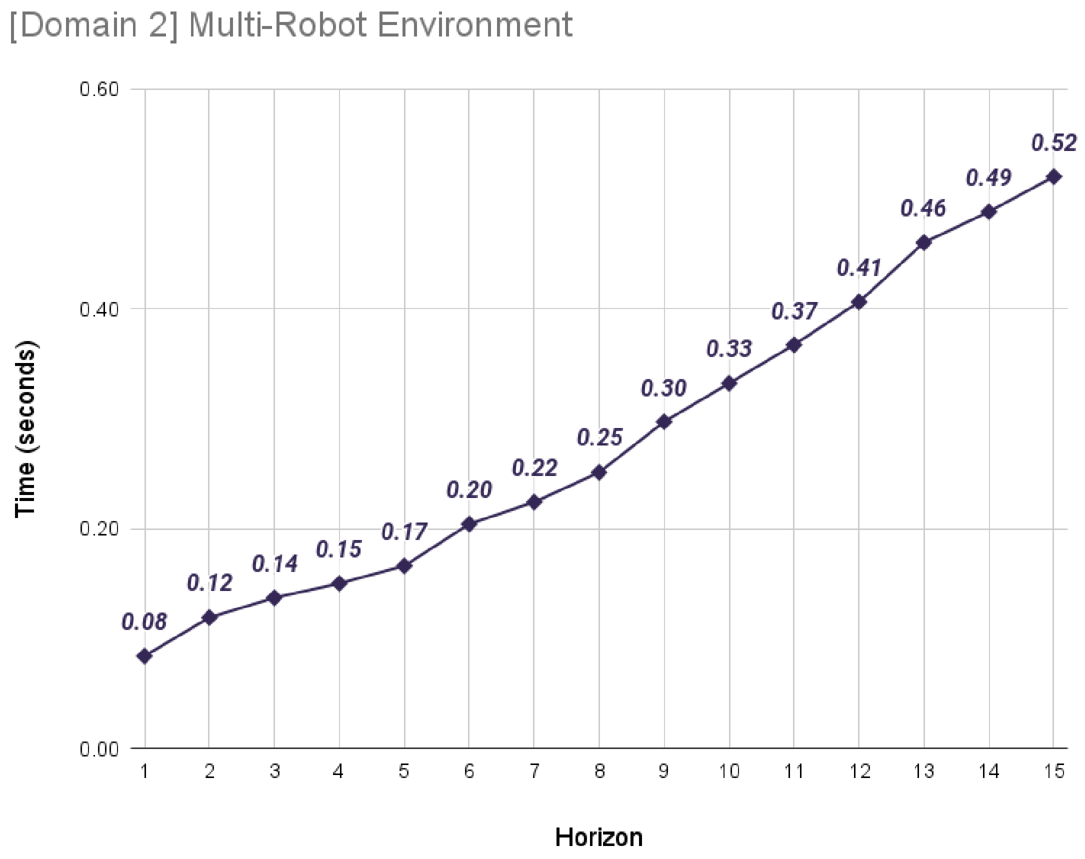


Figure 5.3: Time taken to obtain a valid plan in seconds with respect to the pre-defined plan horizon in multi-robot domain.

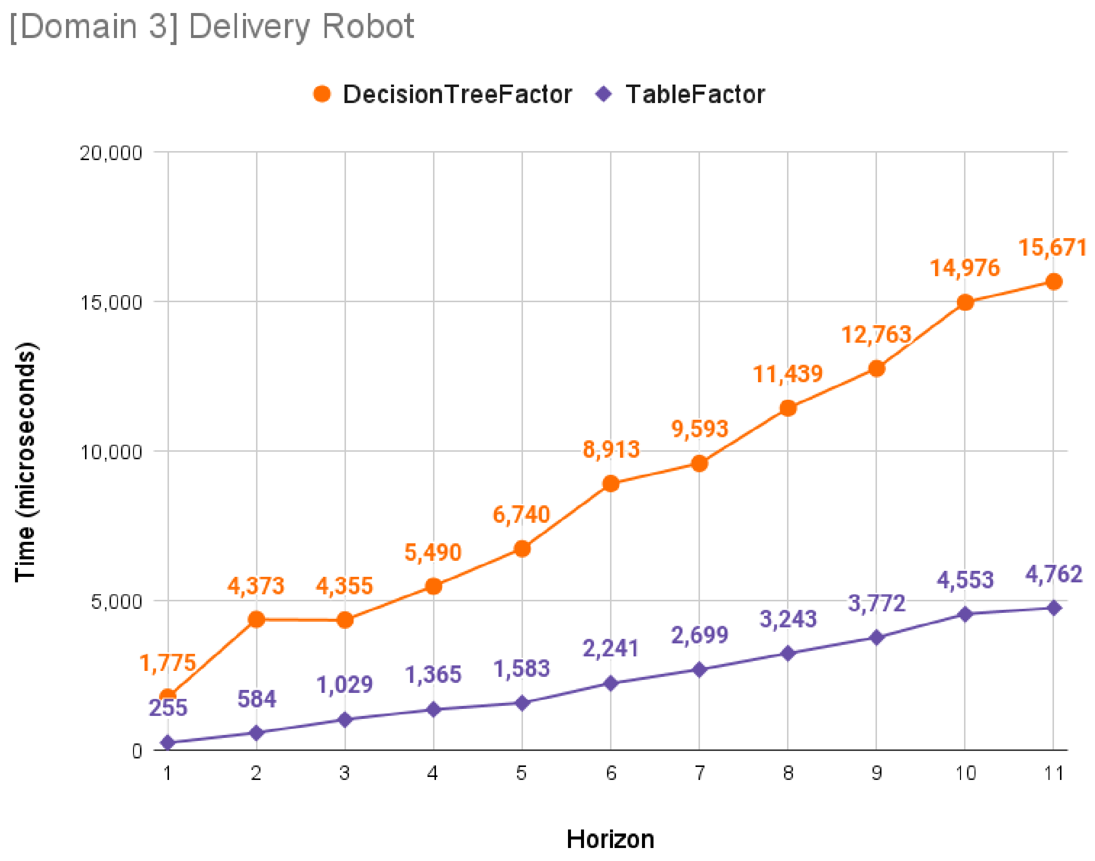


Figure 5.4: Time taken to obtain a valid plan in microseconds with respect to the pre-defined plan horizon in delivery robot domain.

CHAPTER 6
DICUSSION

This thesis represents the classical planning problems in SAS+ formulation in factor graphs. To allow concurrent plans to be found, the binary action variables from the SAS+ formulation are grouped into multi-valued variables.

To alleviate the exponential increase in discrete states, we developed a discrete probabilistic factor optimized for sparsity. The algorithm to multiply the two sparse factors had a runtime of $O(nnz_{f1} \times M_{union} \times V + nnz_{f2} \times M_{f2})$, which significantly sped up the execution time.

Along with some benchmarks on classical planning problems, we investigated formulating a centralized multi-robot environment into a planning problem represented by a factor graph.

Although this thesis focuses on fully observable and deterministic problems, the factor graph representation is capable of partially observable and stochastic problems. In the future, the work done in this thesis can extend to solving a plan with the "highest likelihood of success," given each action has a probability of success.

Also, the generality of the factor graph suggests tight integration perception and planning stacks. A factor graph is already used in perception problems such as SfM and SLAM, and there are works that detect a failure of the perception stack, such as Yang et al. 2021. Therefore, it should be possible to integrate the two stacks and allow planners to plan for better perception while satisfying the goal.

Finally, since formulating classical planning problems in factor graph is shown to be possible in this thesis, and there are works like Mukadam et al. 2018 that uses factor graph to solve continuous motion planning problems, combining the two to create a task and motion planner seems to be a promising next step. There are already works such as Hsiao et al. 2019 and Doherty et al. 2022, which solve discrete-continuous optimization problems on hybrid factor graphs with discrete and continuous variables.

REFERENCES

Dellaert, Frank (2021). “Factor graphs: Exploiting structure in robotics”. In: *Annual Review of Control, Robotics, and Autonomous Systems* 4, pp. 141–166.

Dellaert, Frank, Michael Kaess, et al. (2017a). “Factor graphs for robot perception”. In: *Foundations and Trends® in Robotics* 6.1-2, pp. 1–139.

Baid, Ayush et al. (2021). *GTSFM: Georgia Tech Structure from Motion*. https://github.com/borglab/gtsfm.

Mukadam, Mustafa et al. (2018). “Continuous-time Gaussian process motion planning via probabilistic inference”. In: *The International Journal of Robotics Research* 37.11, pp. 1319–1340.

Mukadam, Mustafa et al. (2019). “STEAP: simultaneous trajectory estimation and planning”. In: *Autonomous Robots* 43.2, pp. 415–434.

Pöschmann, Johannes, Tim Pfeifer, and Peter Protzel (2020). “Factor graph based 3d multi-object tracking in point clouds”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 10343–10350.

Garrett, Caelan Reed et al. (2021). “Integrated task and motion planning”. In: *Annual review of control, robotics, and autonomous systems* 4, pp. 265–293.

Dellaert, Frank and Chris Beall (2017b). “GTSAM 4.0”. In: *URL: https://bitbucket. org/gt-borg/gtsam*.

Doherty, Kevin J. et al. (2022). “Discrete-Continuous Smoothing and Mapping”. In: *IEEE Robotics and Automation Letters* 7.4, pp. 12395–12402.

Hsiao, Ming and Michael Kaess (2019). “Mh-isam2: Multi-hypothesis isam using bayes tree and hypo-tree”. In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 1274–1280.

Fikes, Richard E and Nils J Nilsson (1971). “STRIPS: A new approach to the application of theorem proving to problem solving”. In: *Artificial intelligence* 2.3-4, pp. 189–208.

Nilsson, Nils J et al. (1984). “Shakey the robot”. In:

Bäckström, Christer and Bernhard Nebel (1995). “Complexity results for SAS+ planning”. In: *Computational Intelligence* 11.4, pp. 625–655.

Jonsson, Peter and Christer Bäckström (1998). “State-variable planning under structural restrictions: Algorithms and complexity”. In: *Artificial Intelligence* 100.1-2, pp. 125–176.

Helmert, Malte (2006). “The fast downward planning system”. In: *Journal of Artificial Intelligence Research* 26, pp. 191–246.

Aeronautiques, Constructions et al. (1998). *PDDL— The Planning Domain Definition Language*. Tech. rep. Technical Report.

Fox, Maria and Derek Long (2003). “PDDL2. 1: An extension to PDDL for expressing temporal planning domains”. In: *Journal of artificial intelligence research* 20, pp. 61–124.

Edelkamp, Stefan and Jörg Hoffmann (2004). *PDDL2. 2: The language for the classical part of the 4th international planning competition*. Tech. rep. Technical Report 195, University of Freiburg.

Gerevini, Alfonso and Derek Long (2005). “Plan constraints and preferences in PDDL3-the language of the fifth international planning competition”. In:

Gent, Ian P et al. (2007). “Data structures for generalised arc consistency for extensional constraints”. In: *AAAI*. Vol. 7, pp. 191–197.

Vidal, Vincent and Héctor Geffner (2006). “Branching and pruning: An optimal temporal POCL planner based on constraint programming”. In: *Artificial Intelligence* 170.3, pp. 298–335.

Barták, Roman and Daniel Toropila (2008). “Reformulating Constraint Models for Classical Planning.” In: *FLAIRS Conference*, pp. 525–530.

Lopez, Adriana and Fahiem Bacchus (2003). “Generalizing graphplan by formulating planning as a CSP”. In: *IJCAI*. Vol. 3, pp. 954–960.

Helmert, Malte (2009). “Concise finite-domain representations for PDDL planning tasks”. In: *Artificial Intelligence* 173.5-6, pp. 503–535.

Gustavson, Fred G (1972). “Some basic techniques for solving sparse systems of linear equations”. In: *Sparse matrices and their applications*. Springer, pp. 41–52.

Guennebaud, Gaël, Benoît Jacob, et al. (2010). *Eigen v3*. http://eigen.tuxfamily.org.

Seipp, Jendrik et al. (2016). “Correlation complexity of classical planning domains”. In:

Francès Medina, Guillem et al. (2019). “Generalized potential heuristics for classical planning”. In: International Joint Conferences on Artificial Intelligence.

Patrizi, Fabio et al. (2011). “Computing infinite plans for LTL goals using a classical planner”. In: *Twenty-Second International Joint Conference on Artificial Intelligence*.

Poole, David L and Alan K Mackworth (2010). *Artificial Intelligence: foundations of computational agents*. Cambridge University Press.

Yang, Heng and Luca Carlone (2021). “Certifiable outlier-robust geometric perception: exact semidefinite relaxations and scalable global optimization”. In: *arXiv preprint arXiv:2109.03349*.