CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Decision Analytics

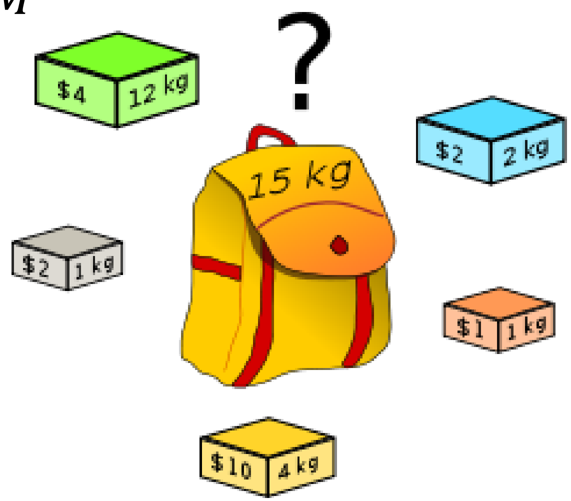
Lecture 2: OR Tools

Example: Knapsack problem

Given a knapsack with a capacity to hold M kg and a set of items with weights $w_1, \ldots, w_n$ and a value $v_1, \ldots, v_n$, what is the maximum value we can carry without exceeding the capacity limit.

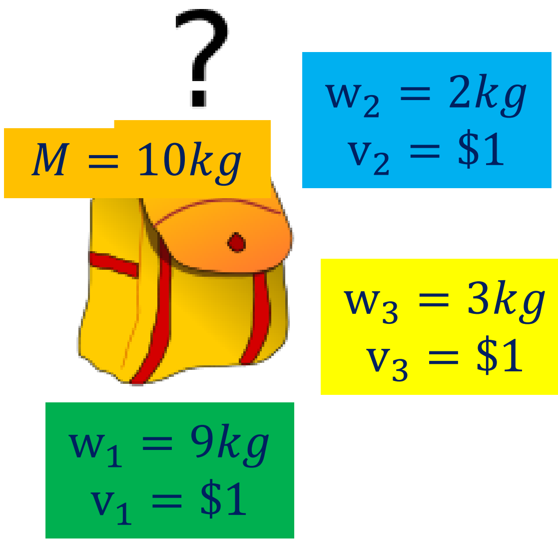
More formal we look for $x_i \in \{0,1\}$ to

- Maximise the value $\mathrm{V} = \sum x_i v_i$
- Subject to the capacity constraint $\mathrm{W} = \sum x_i w_i \leq M$



Example: Knapsack problem

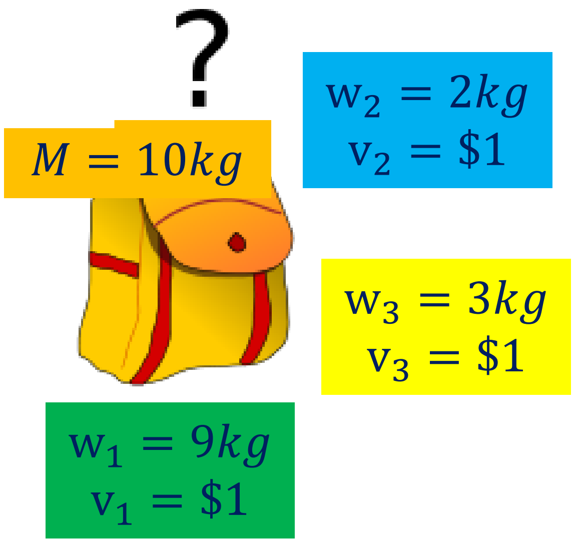
So how can we solve this problem?



Example: Knapsack problem

So how can we solve this problem?

- We could start with an empty bag $\mathbf{x} = [0,0,0]$
- This initial state is valid, as the weight is $\mathrm{W} = 0 \leq 10$

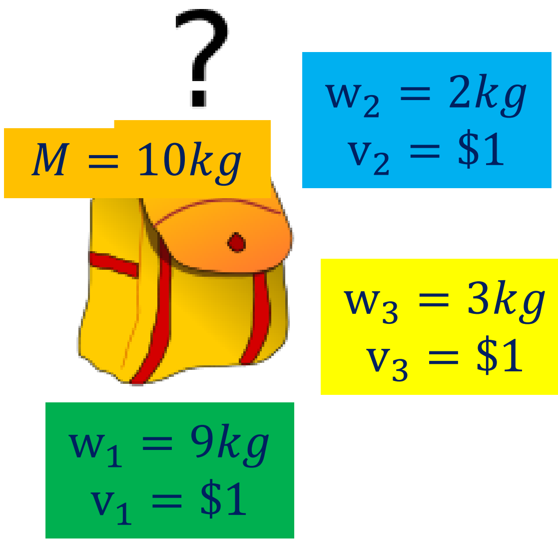


Example: Knapsack problem

So how can we solve this problem?

- We could start with an empty bag $\mathbf{x} = [0,0,0]$
- This initial state is valid, as the weight is $\mathrm{W} = 0 \leq 10$

- We could start with adding the first item, i.e. $\mathbf{x} = [1,0,0]$
- This is a valid state, as the weight is now $\mathrm{W} = 9 \leq 10$
- We can no longer add items, as every other item would exceed the capacity limit and create an invalid state
- The value of this final state is $\mathrm{V} = 1$



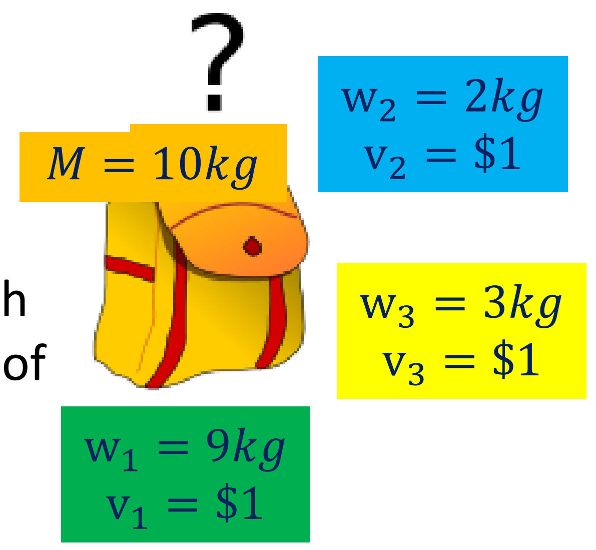
Example: Knapsack problem

So how can we solve this problem?

- We could start with an empty bag $\mathbf{x} = [0,0,0]$
- This initial state is valid, as the weight is $\mathrm{W} = 0 \leq 10$

- We could start with adding the first item, i.e. $\mathbf{x} = [1,0,0]$
- This is a valid state, as the weight is now $\mathrm{W} = 9 \leq 10$
- We can no longer add items, as every other item would exceed the capacity limit and create an invalid state
- The value of this final state is $\mathrm{V} = 1$

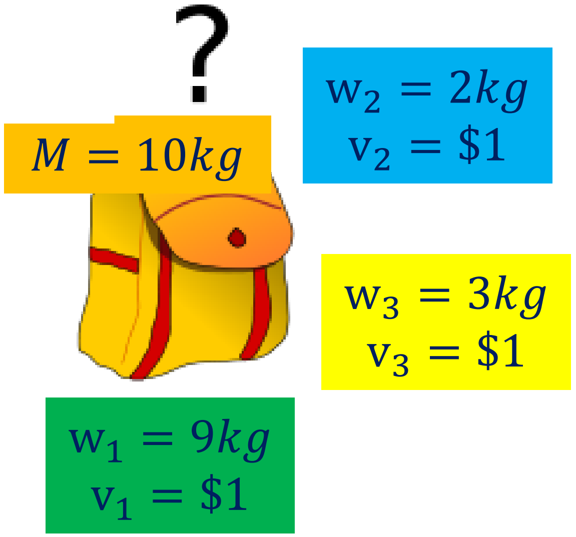
- Apparently we made the **wrong decision**!
- If we had selected the other two items first, i.e. $\mathbf{x} = [0,1,1]$, then we would have achieved a valid state with bag weight $\mathrm{W} = 5 \leq 10$ and crucially a content value of $\mathrm{V} = 2$
- This is the best state achievable



Example: Knapsack problem

Let's look at the problem again

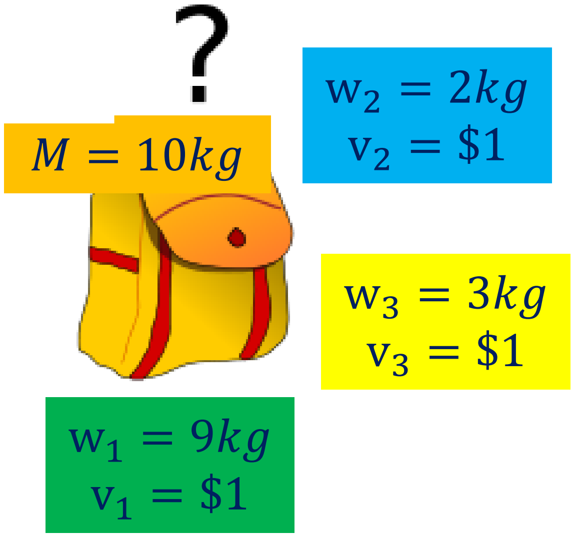
- There are three binary variables, therefore $2^3 = 8$ states



Example: Knapsack problem

Let’s look at the problem again

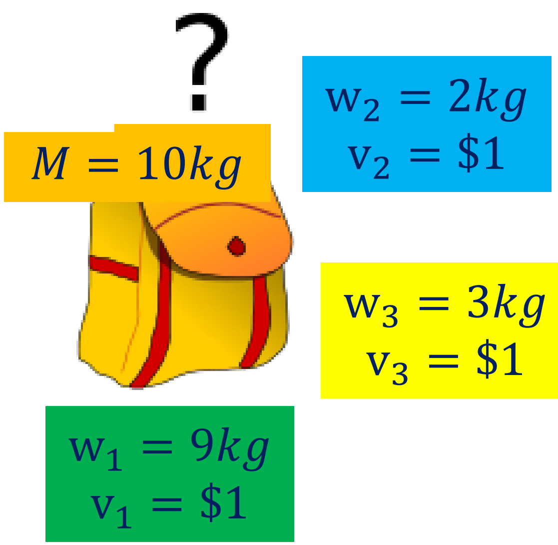
- There are three binary variables, therefore $2^3 = 8$ states
  - 5 **valid states**, where $W \leq 10$
    - $\boldsymbol{x} = [0,0,0] \Rightarrow W = 0, V = 0$
    - $\boldsymbol{x} = [0,0,1] \Rightarrow W = 3, V = 1$
    - $\boldsymbol{x} = [0,1,0] \Rightarrow W = 2, V = 1$
    - $\boldsymbol{x} = [0,1,1] \Rightarrow W = 5, V = 2$
    - $\boldsymbol{x} = [1,0,0] \Rightarrow W = 9, V = 1$



Example: Knapsack problem

Let's look at the problem again

- There are three binary variables, therefore $2^3 = 8$ states
  - 5 **valid states**, where $W \le 10$
    - $\boldsymbol{x} = [0,0,0] \Rightarrow W = 0, V = 0$
    - $\boldsymbol{x} = [0,0,1] \Rightarrow W = 3, V = 1$
    - $\boldsymbol{x} = [0,1,0] \Rightarrow W = 2, V = 1$
    - $\boldsymbol{x} = [0,1,1] \Rightarrow W = 5, V = 2$
    - $\boldsymbol{x} = [1,0,0] \Rightarrow W = 9, V = 1$
  - 3 **invalid states**, where $W > 10$
    - $\boldsymbol{x} = [1,0,1] \Rightarrow W = 12, V = 2$
    - $\boldsymbol{x} = [1,1,0] \Rightarrow W = 11, V = 2$
    - $\boldsymbol{x} = [1,1,1] \Rightarrow W = 14, V = 3$

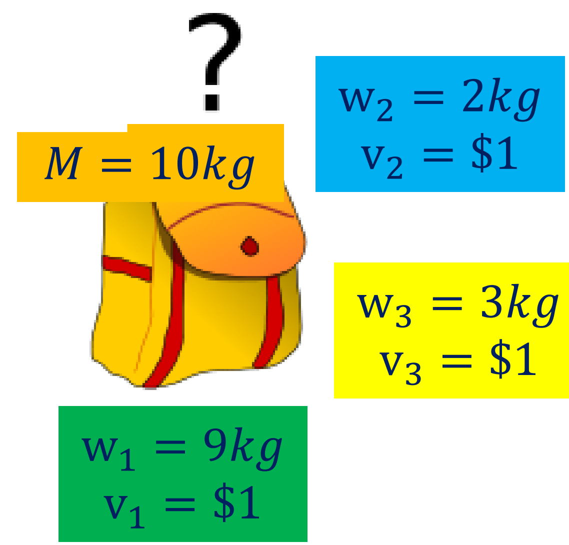


Example: Knapsack problem

Let's look at the problem again

- There are three binary variables, therefore $2^3 = 8$ states
  - 5 **valid states**, where $W \leq 10$
    - $\boldsymbol{x} = [0,0,0] \Rightarrow W = 0, V = 0$
    - $\boldsymbol{x} = [0,0,1] \Rightarrow W = 3, V = 1$
    - $\boldsymbol{x} = [0,1,0] \Rightarrow W = 2, V = 1$
    - $\boldsymbol{x} = [0,1,1] \Rightarrow W = 5, V = 2$
    - $\boldsymbol{x} = [1,0,0] \Rightarrow W = 9, V = 1$
  - 3 **invalid states**, where $W > 10$
    - $\boldsymbol{x} = [1,0,1] \Rightarrow W = 12, V = 2$
    - $\boldsymbol{x} = [1,1,0] \Rightarrow W = 11, V = 2$
    - $\boldsymbol{x} = [1,1,1] \Rightarrow W = 14, V = 3$

1 best state



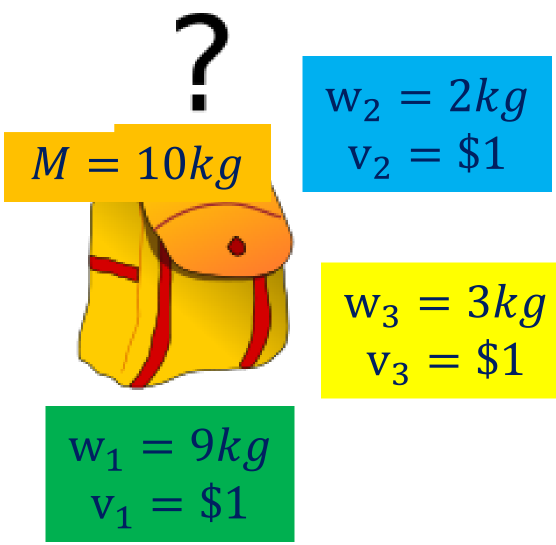
Example: Knapsack problem

Let's look at the problem again

- There are three binary variables, therefore $2^3 = 8$ states
  - 5 **valid states**, where $W \leq 10$
    - $\boldsymbol{x} = [0,0,0] \Rightarrow W = 0, V = 0$
    - $\boldsymbol{x} = [0,0,1] \Rightarrow W = 3, V = 1$
    - $\boldsymbol{x} = [0,1,0] \Rightarrow W = 2, V = 1$
    - $\boldsymbol{x} = [0,1,1] \Rightarrow W = 5, V = 2$
    - $\boldsymbol{x} = [1,0,0] \Rightarrow W = 9, V = 1$
  - 3 **invalid states**, where $W > 10$
    - $\boldsymbol{x} = [1,0,1] \Rightarrow W = 12, V = 2$
    - $\boldsymbol{x} = [1,1,0] \Rightarrow W = 11, V = 2$
    - $\boldsymbol{x} = [1,1,1] \Rightarrow W = 14, V = 3$

1 best state

- But how do we explore this space systematically?

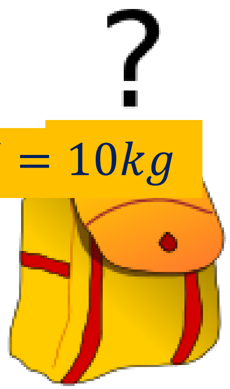


Example: Knapsack problem

Starting from the empty bag we have four options (decisions):

$\boldsymbol{x} = [0,0,0]$

?



$M = 10kg$

$w_2 = 2kg$
$v_2 = \$1$

$w_3 = 3kg$
$v_3 = \$1$

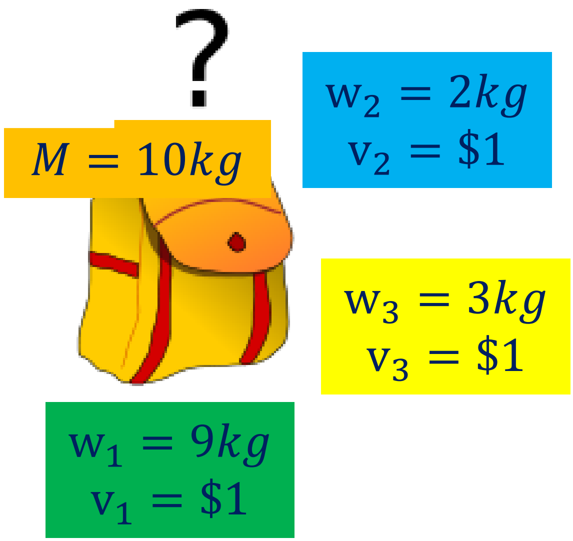
$w_1 = 9kg$
$v_1 = \$1$

Example: Knapsack problem

Starting from the empty bag we have four options (decisions):
- Don’t add anything ⇒ $W = 0, V = 0$

$\boldsymbol{x} = [0,0,0]$

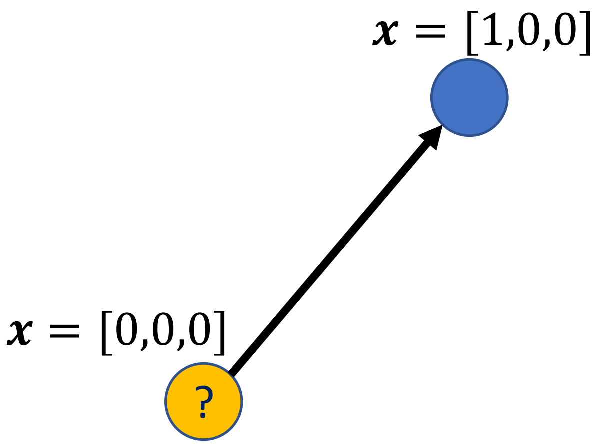
?

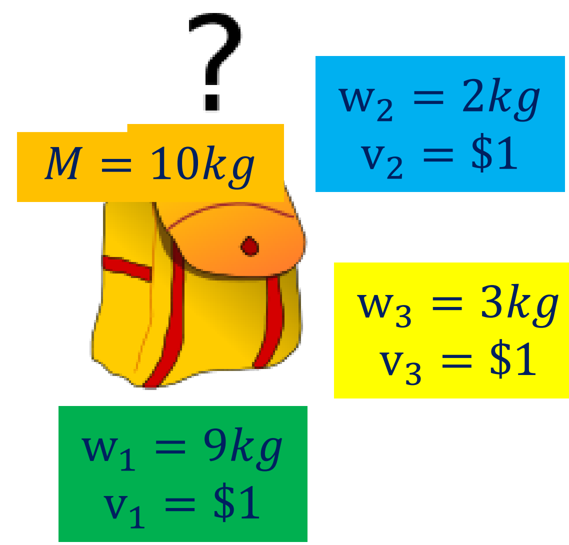


Example: Knapsack problem

Starting from the empty bag we have four options (decisions):

- Don't add anything $\Rightarrow W = 0, V = 0$
- Add first item $\Rightarrow W = 9, V = 1$

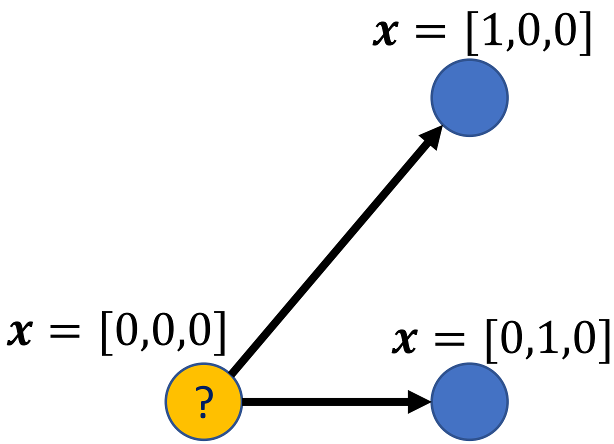


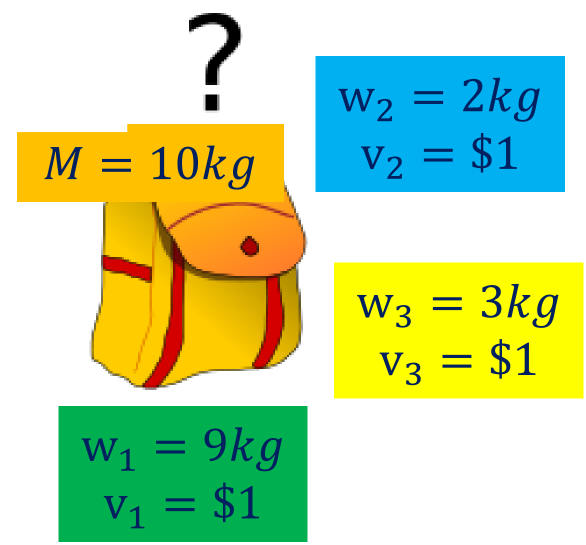


Example: Knapsack problem

Starting from the empty bag we have four options (decisions):

- Don't add anything $\Rightarrow W = 0, V = 0$
- Add first item $\Rightarrow W = 9, V = 1$
- Add second item $\Rightarrow W = 2, V = 1$

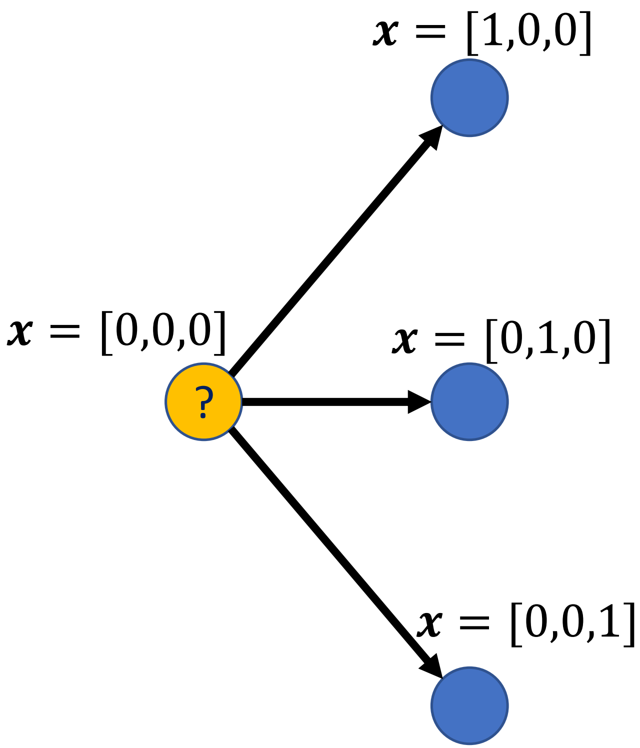


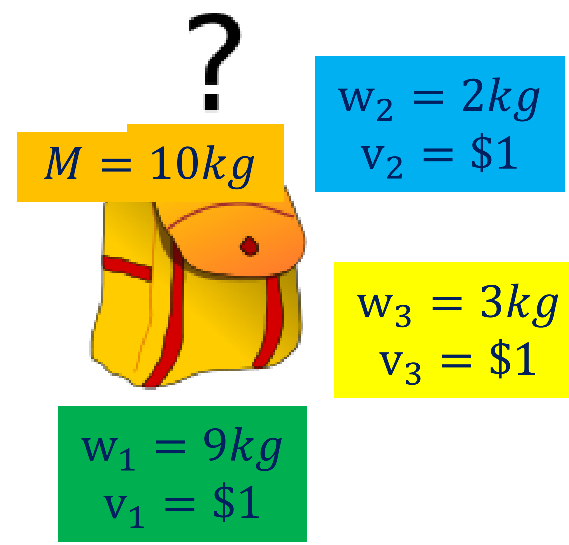


Example: Knapsack problem

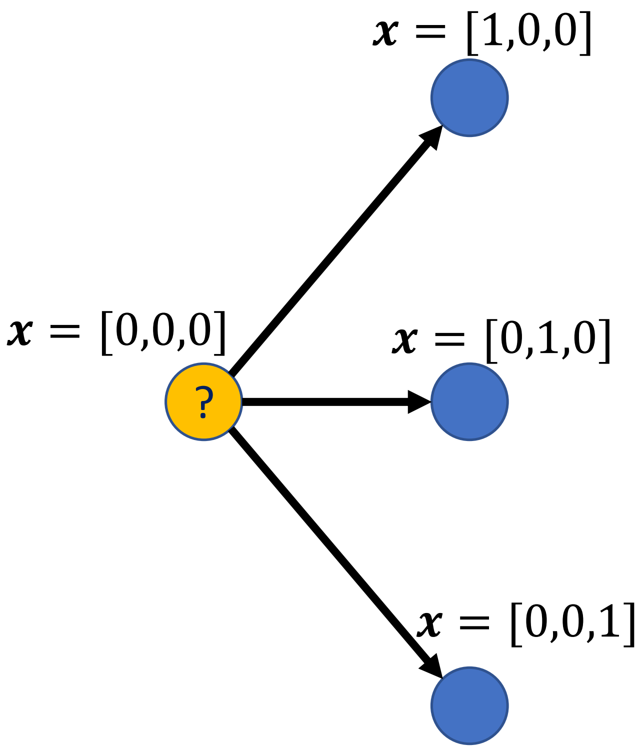
Starting from the empty bag we have four options (decisions):

- Don't add anything $\Rightarrow W = 0, V = 0$
- Add first item $\Rightarrow W = 9, V = 1$
- Add second item $\Rightarrow W = 2, V = 1$
- Add third item $\Rightarrow W = 2, V = 1$





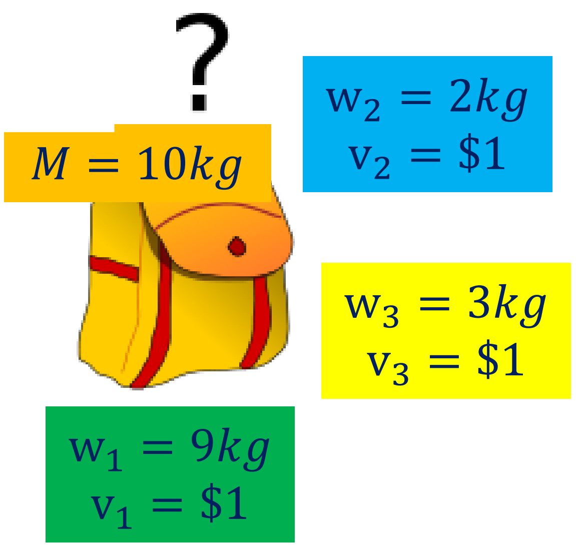
Example: Knapsack problem



Starting from the empty bag we have four options (decisions):

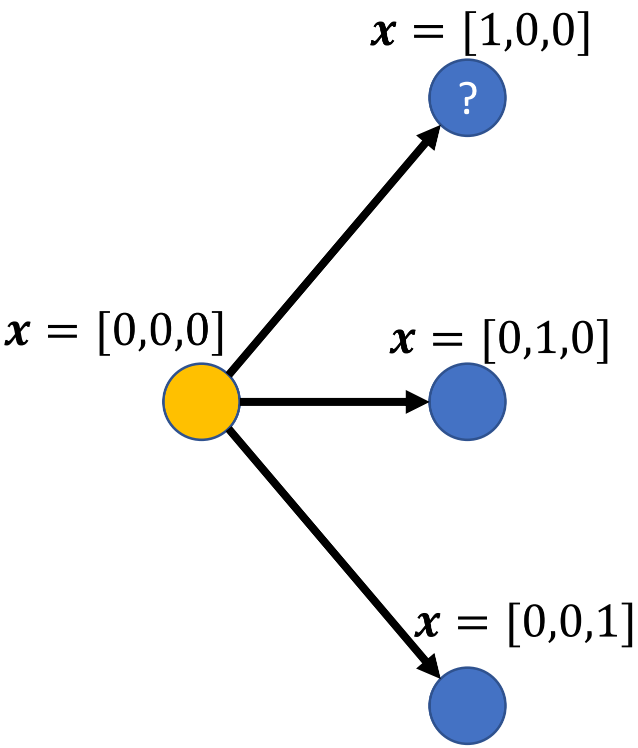
- Don't add anything $\Rightarrow W = 0, V = 0$
- Add first item $\Rightarrow W = 9, V = 1$
- Add second item $\Rightarrow W = 2, V = 1$
- Add third item $\Rightarrow W = 2, V = 1$

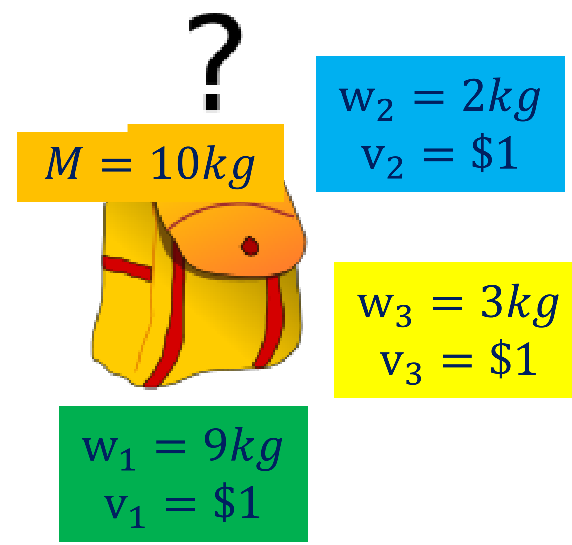
- All four states are valid, and hence solutions for the knapsack problem
- But, which of these decisions is the best?



Example: Knapsack problem

Lets select the first item and see where it leads:

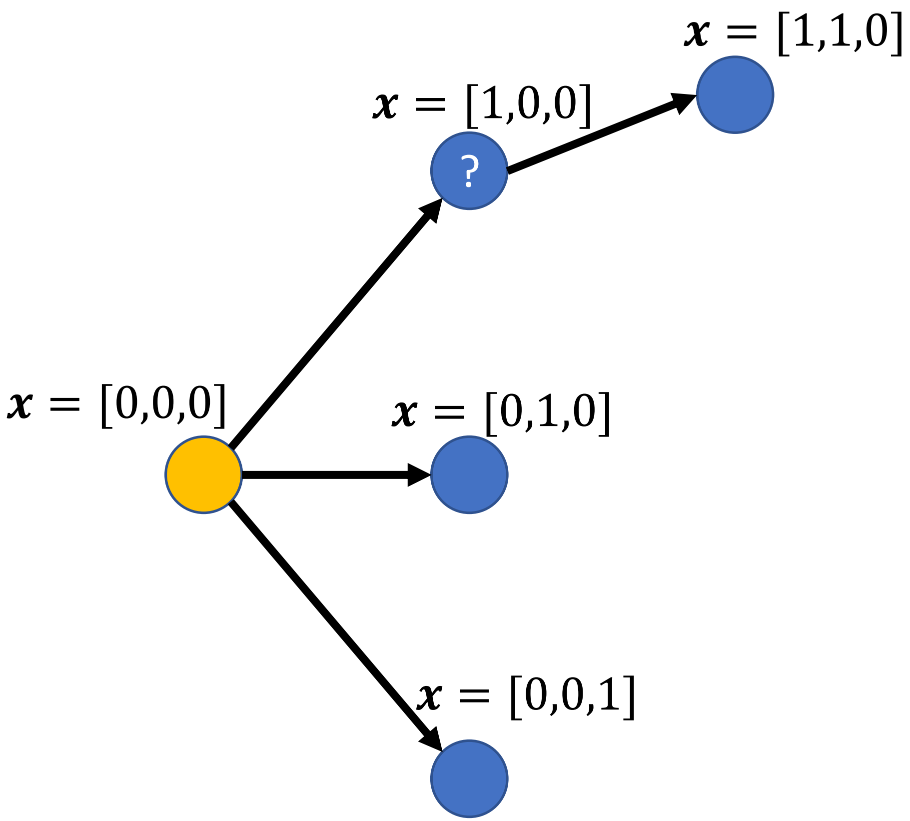


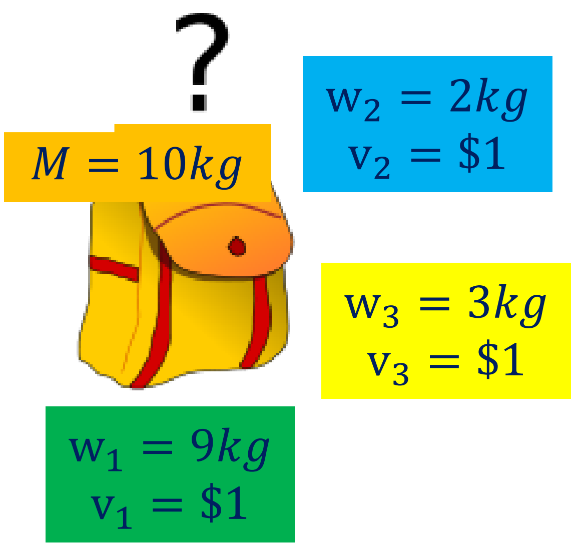


Example: Knapsack problem

Lets select the first item and see where it leads:

- Try to add item two $\Rightarrow W = 11$

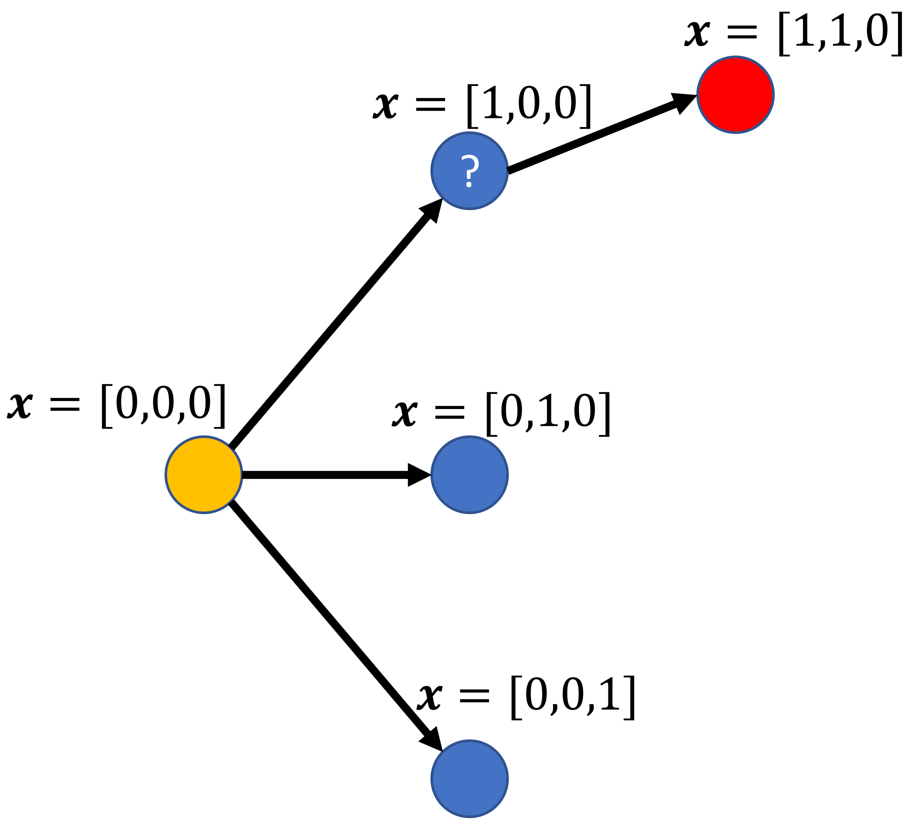


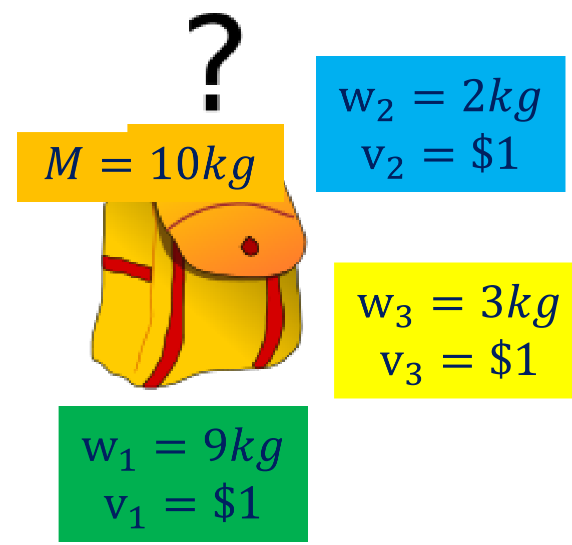


Example: Knapsack problem

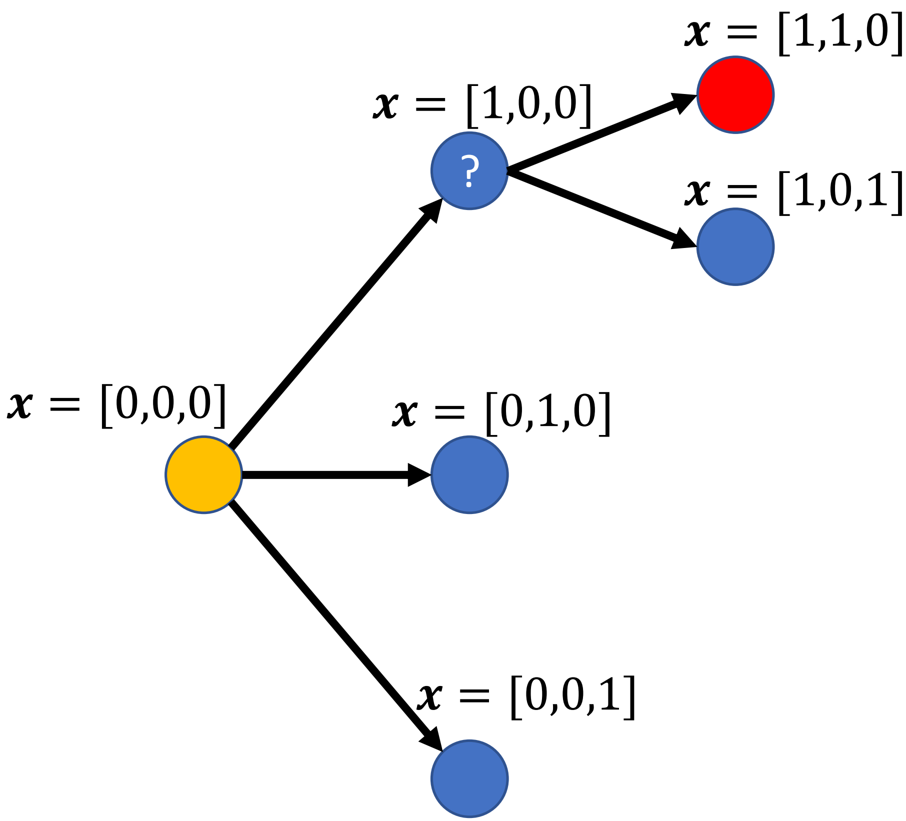
Lets select the first item and see where it leads:

- Try to add item two $\Rightarrow W = 11 \Rightarrow$ invalid



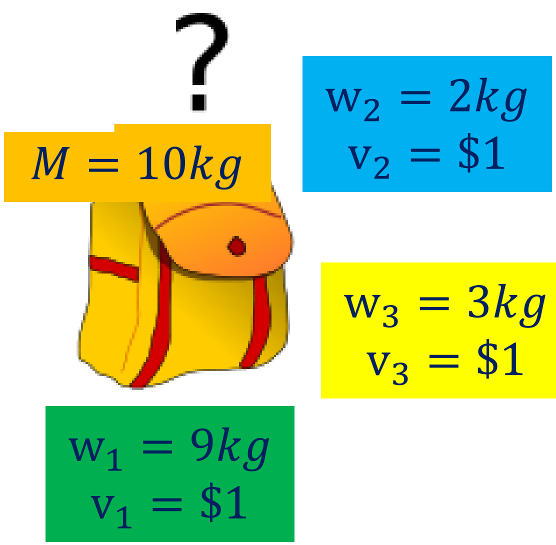


Example: Knapsack problem

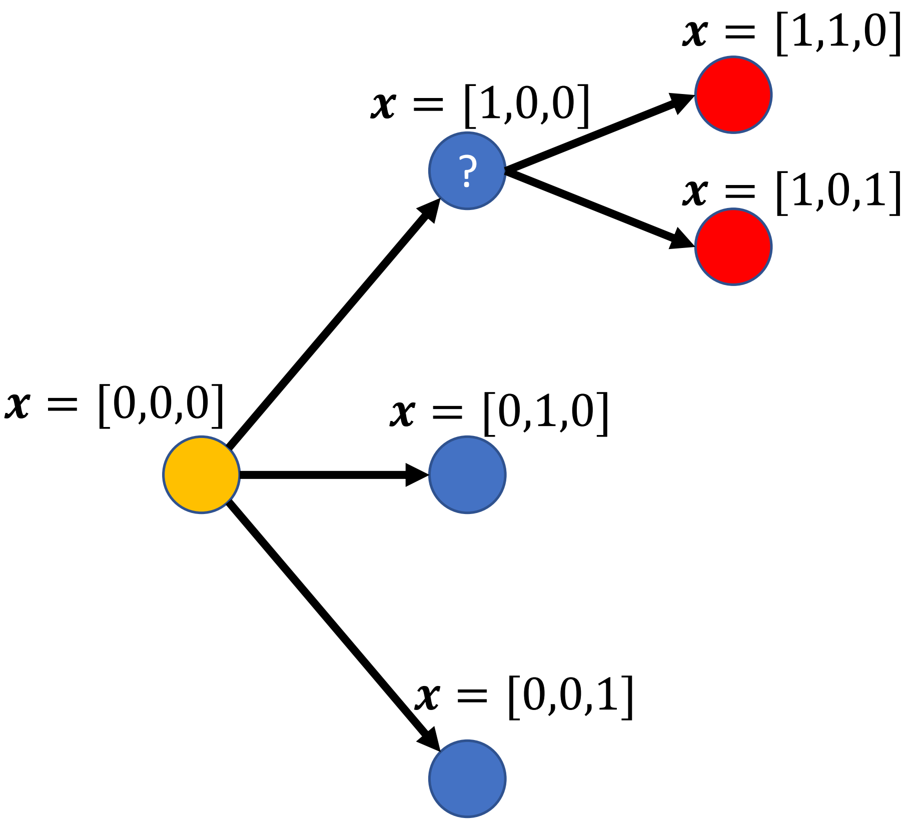


Lets select the first item and see where it leads:

- Try to add item two $\Rightarrow W = 11 \Rightarrow$ invalid
- Try to add item two $\Rightarrow W = 12$

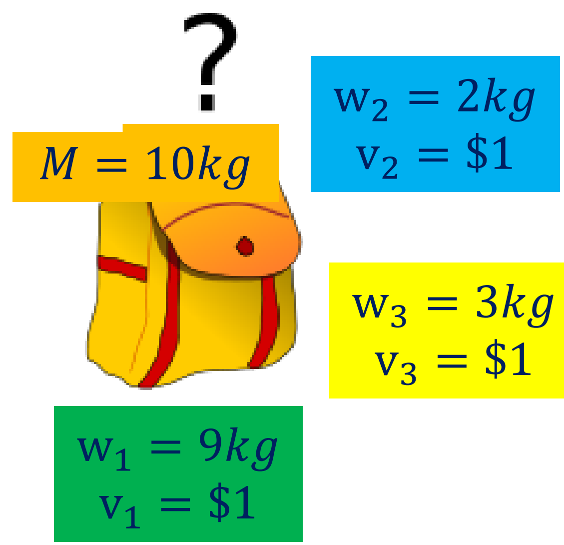


Example: Knapsack problem

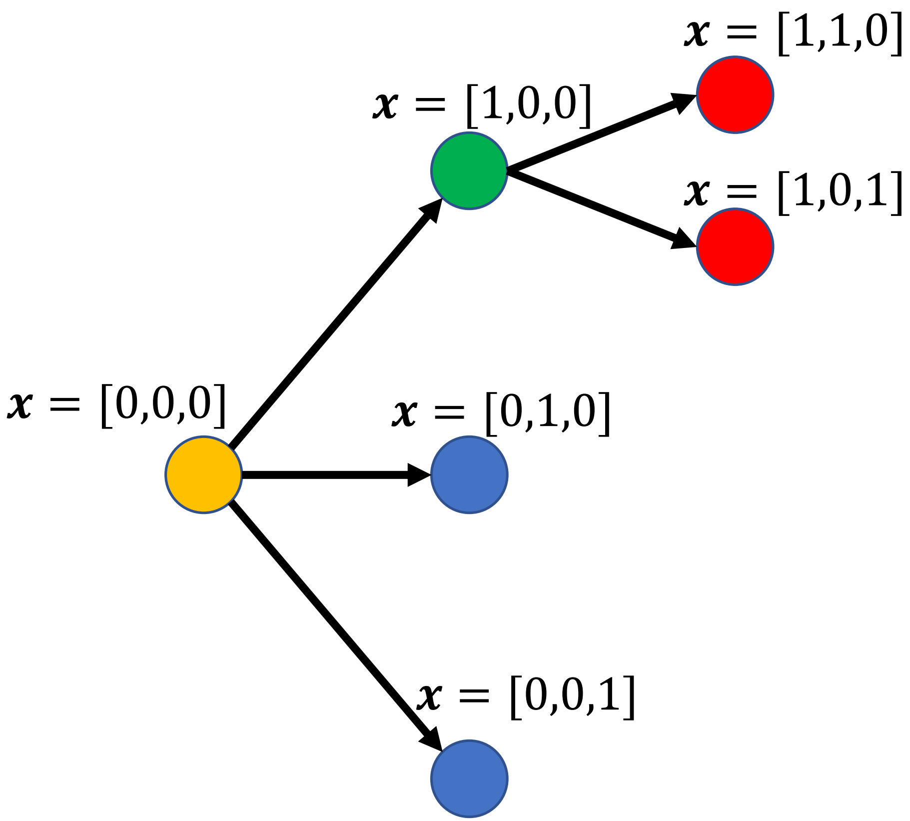


Lets select the first item and see where it leads:

- Try to add item two $\Rightarrow W = 11 \Rightarrow$ invalid
- Try to add item two $\Rightarrow W = 12 \Rightarrow$ invalid



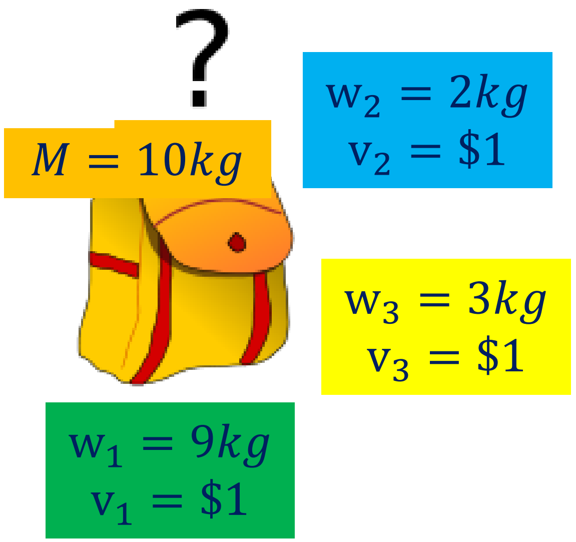
Example: Knapsack problem



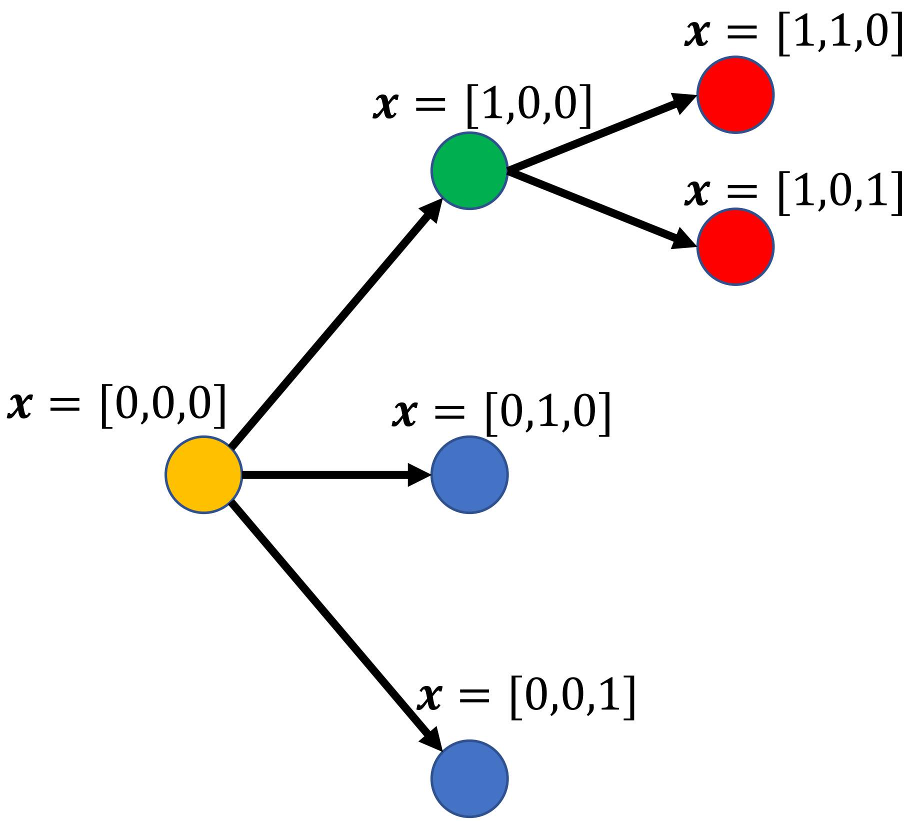
Lets select the first item and see where it leads:

- Try to add item two $\Rightarrow W = 11 \Rightarrow$ invalid
- Try to add item two $\Rightarrow W = 12 \Rightarrow$ invalid

- We have discovered a **final state**



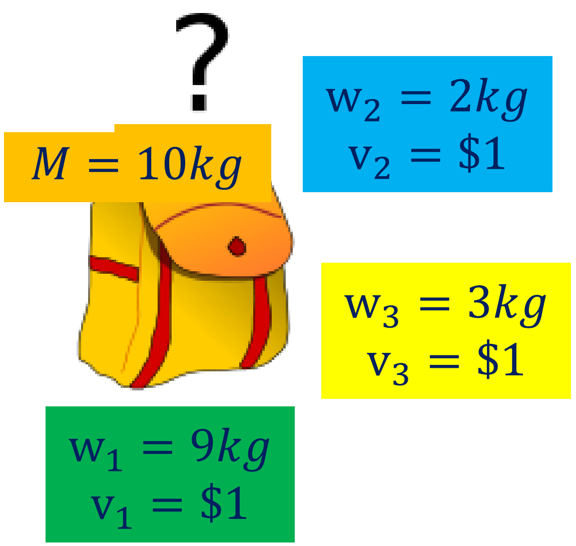
Example: Knapsack problem



Lets select the first item and see where it leads:

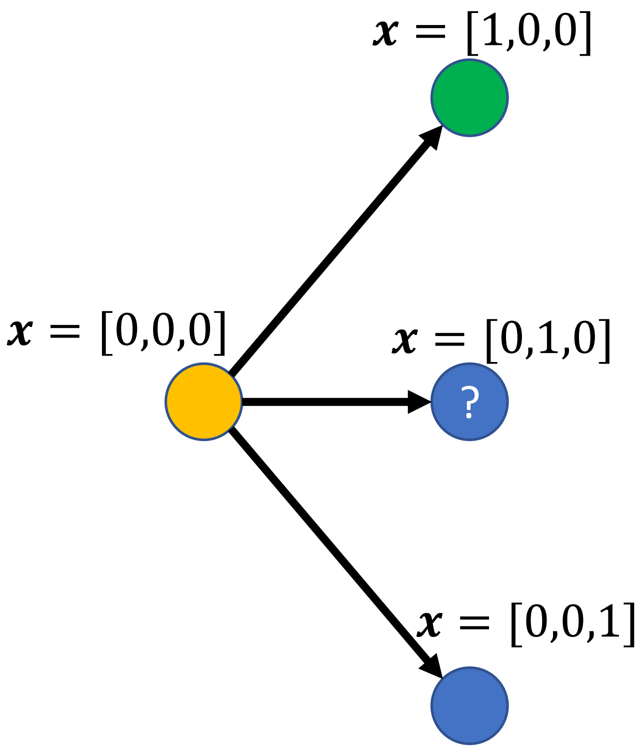
- Try to add item two $\Rightarrow W = 11 \Rightarrow$ invalid
- Try to add item two $\Rightarrow W = 12 \Rightarrow$ invalid

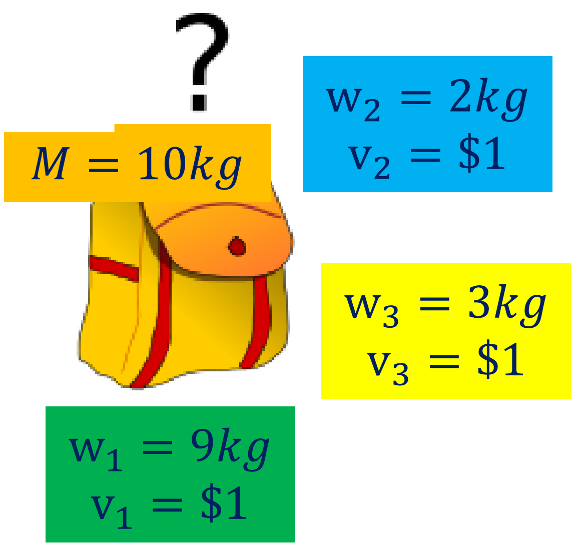
- We have discovered a **final state**
- This is where a "greedy algorithm" terminates, however what if we could backtrack and find a better solution



Example: Knapsack problem

So let's backtrack and select the second item first:

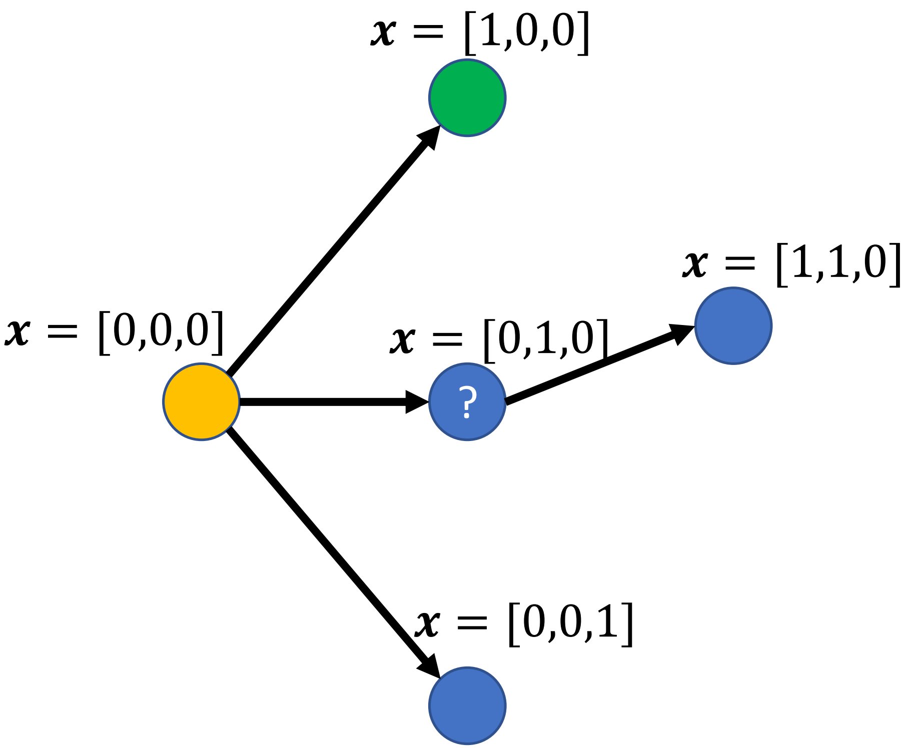


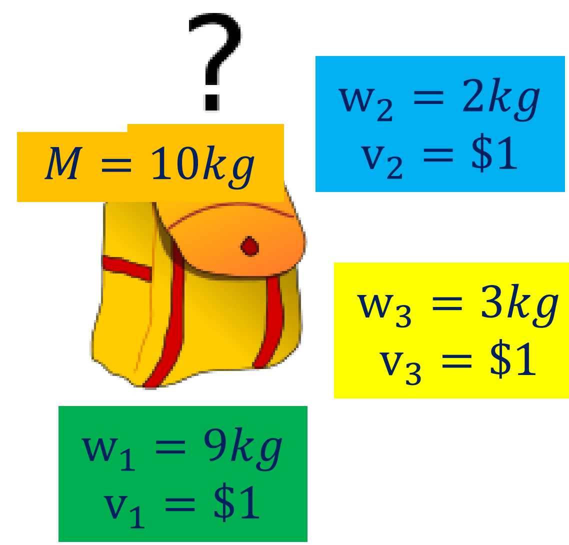


Example: Knapsack problem

So let's backtrack and select the second item first:

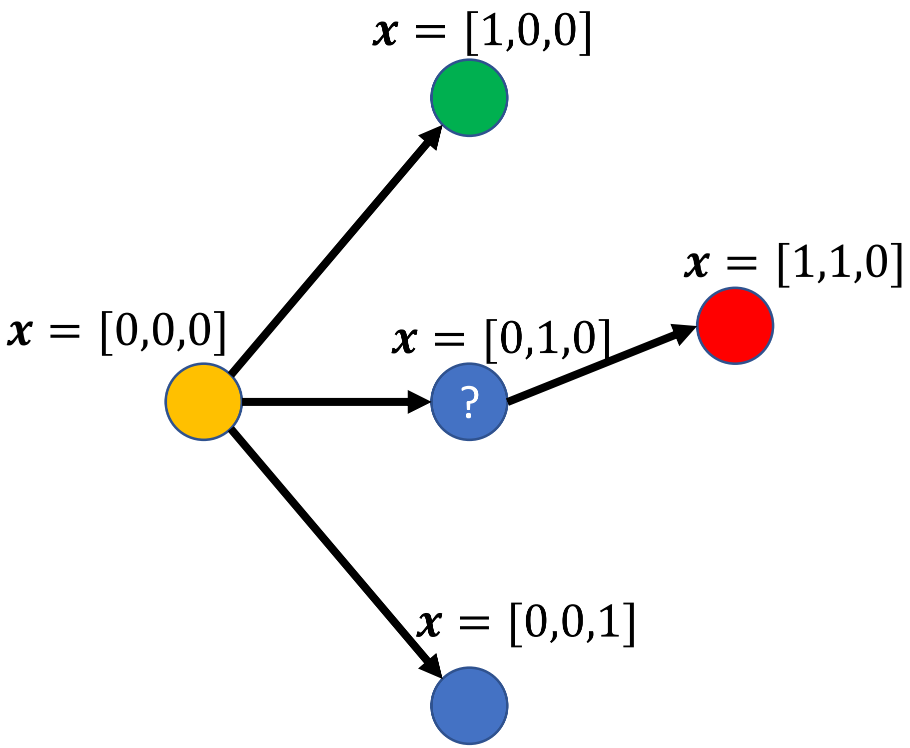
- Try adding item one next ⇒ $W = 11$

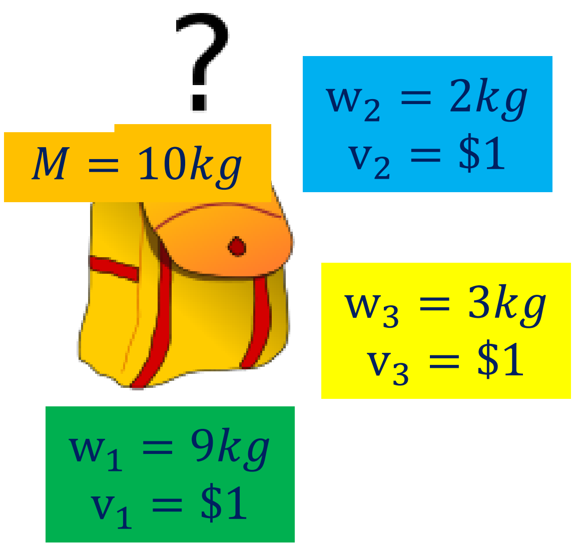




Example: Knapsack problem

So let's backtrack and select the second item first:
- Try adding item one next $\Rightarrow W = 11 \Rightarrow$ invalid

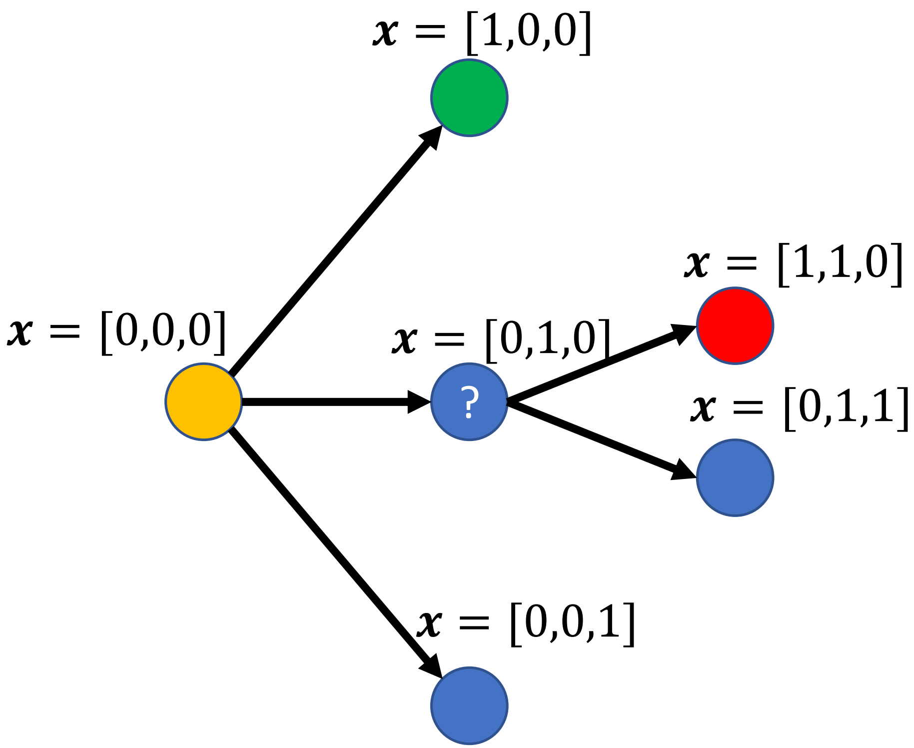


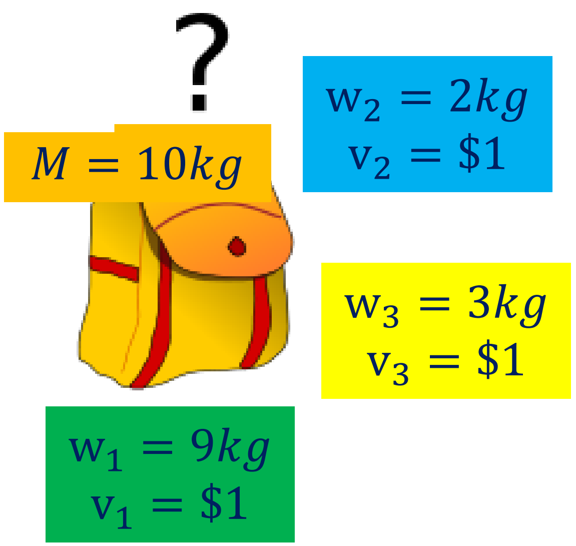


Example: Knapsack problem

So let's backtrack and select the second item first:

- Try adding item one next $\Rightarrow W = 11 \Rightarrow$ invalid
- Try adding item three next $\Rightarrow W = 5$



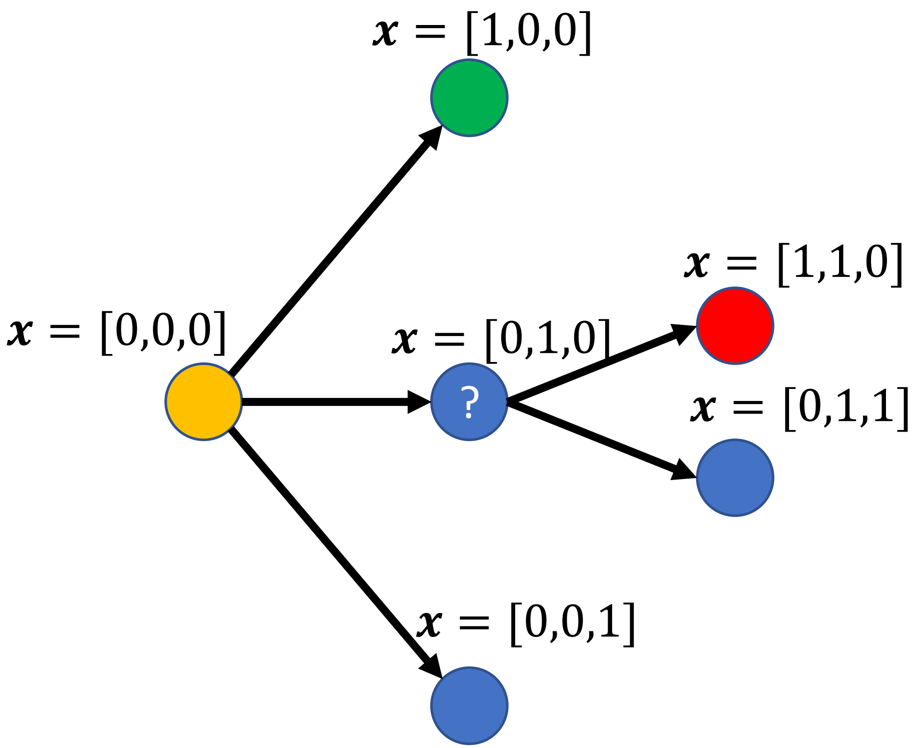


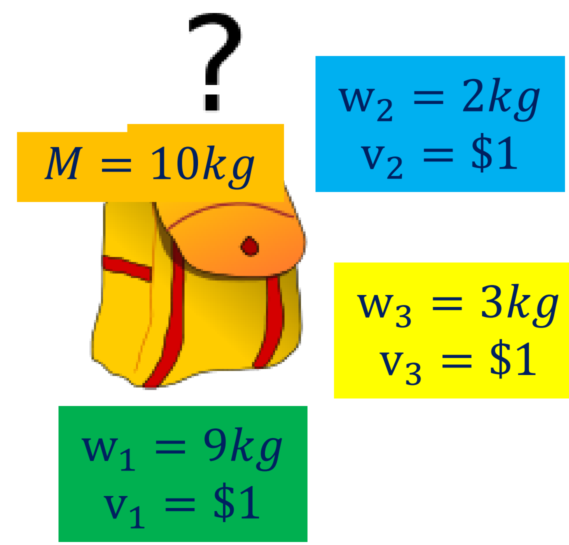
Example: Knapsack problem

So let's backtrack and select the second item first:

- Try adding item one next $\Rightarrow W = 11 \Rightarrow$ invalid
- Try adding item three next $\Rightarrow W = 5$

- We found a solution with $V = 2$



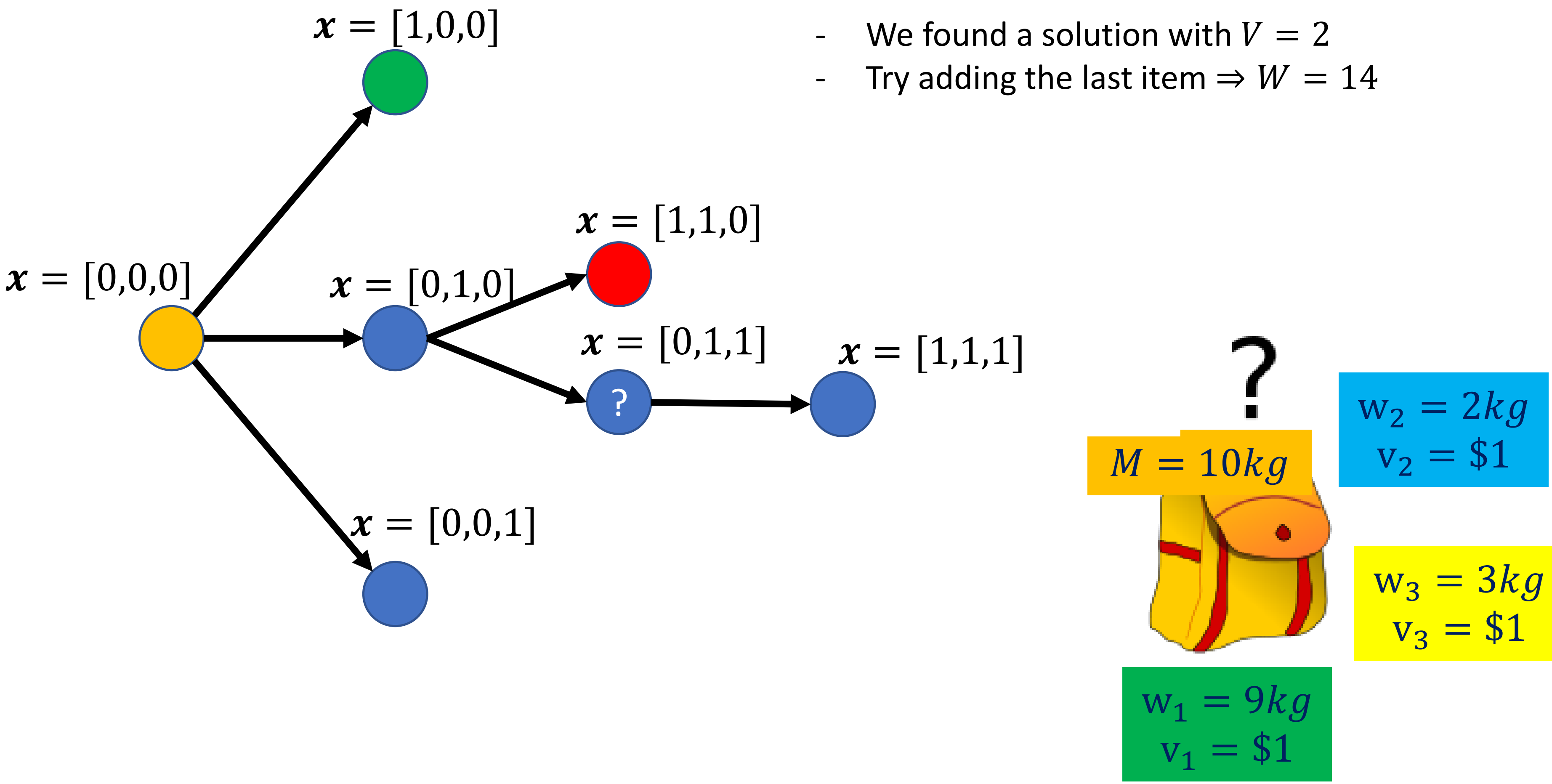


Example: Knapsack problem

So let's backtrack and select the second item first:

- Try adding item one next ⇒ $W = 11$ ⇒ invalid
- Try adding item three next ⇒ $W = 5$

- We found a solution with $V = 2$
- Try adding the last item ⇒ $W = 14$

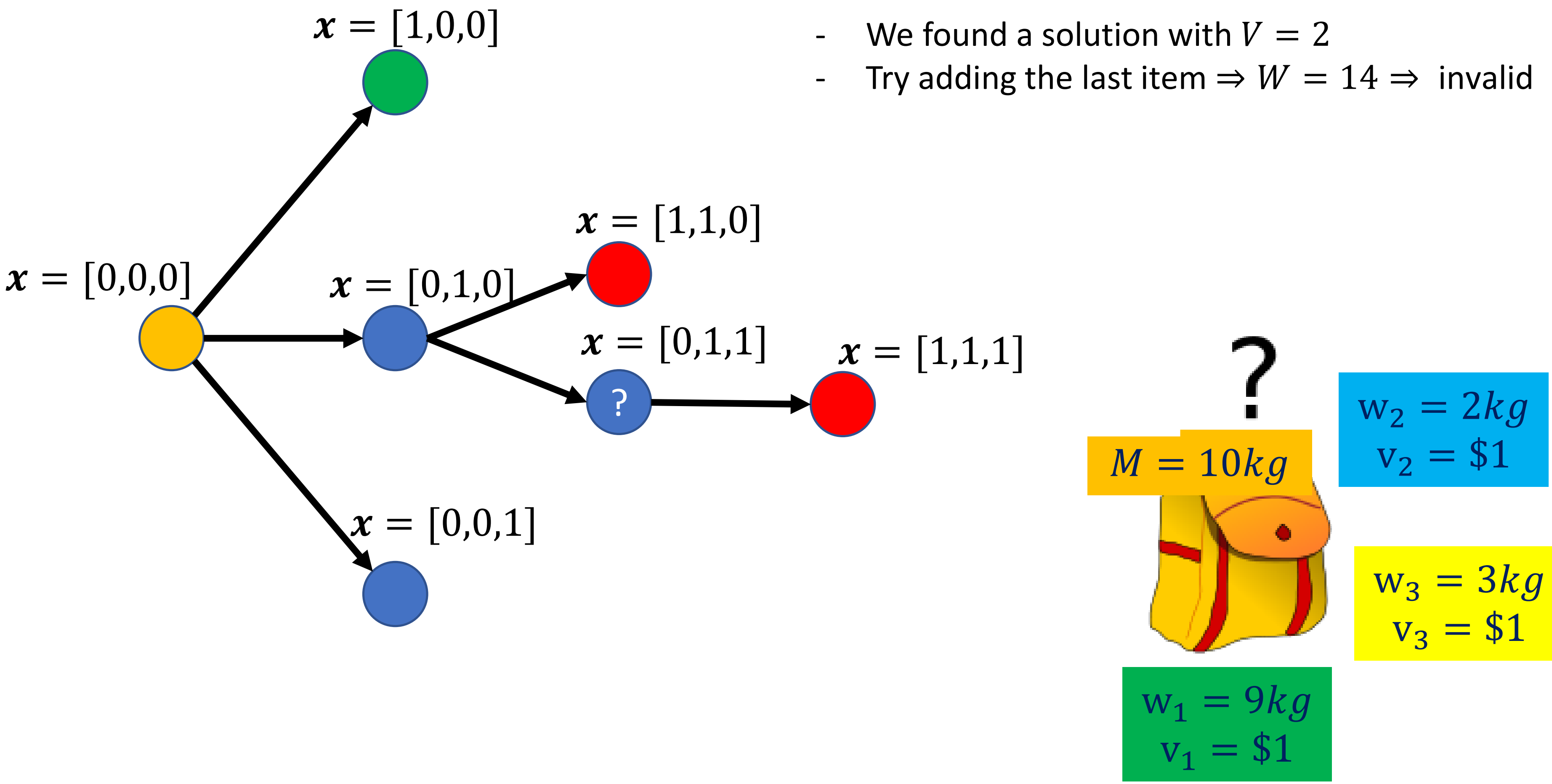


Example: Knapsack problem

So let's backtrack and select the second item first:

- Try adding item one next $\Rightarrow W = 11 \Rightarrow$ invalid
- Try adding item three next $\Rightarrow W = 5$

- We found a solution with $V = 2$
- Try adding the last item $\Rightarrow W = 14 \Rightarrow$ invalid

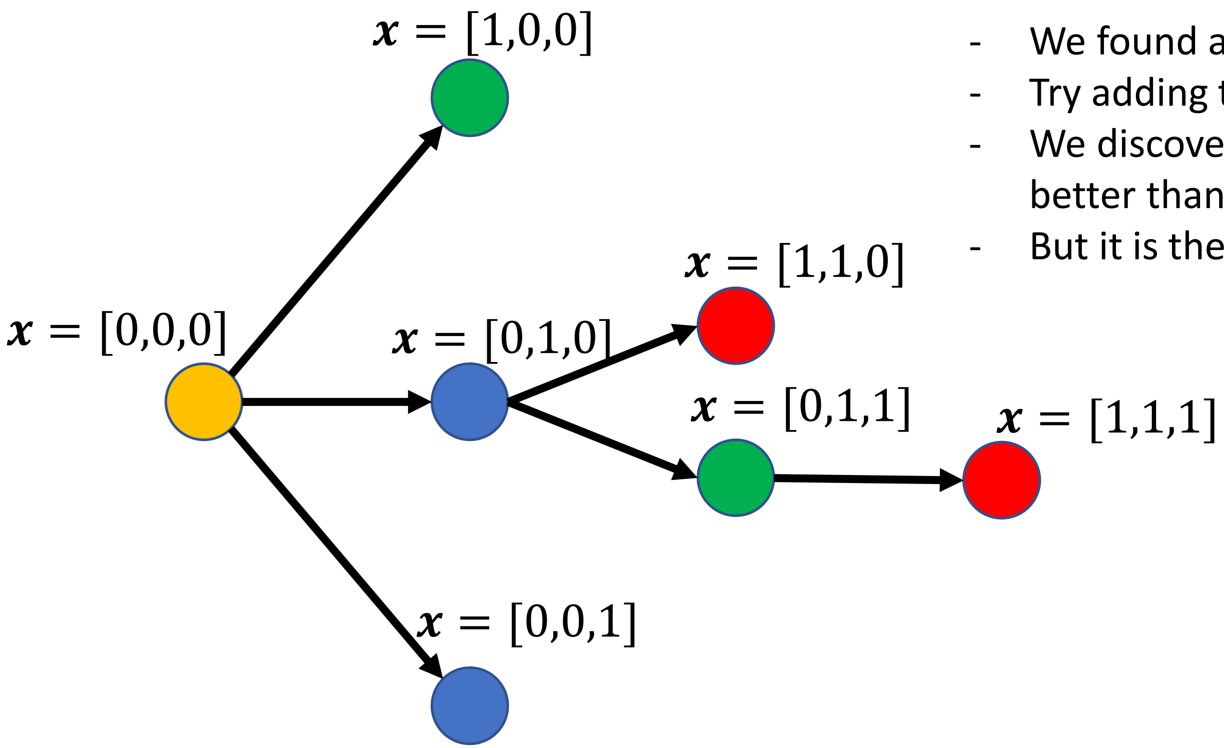


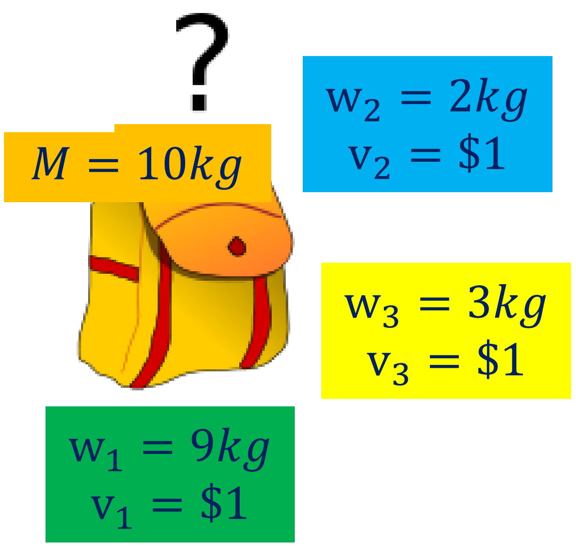
Example: Knapsack problem

So let's backtrack and select the second item first:

- Try adding item one next ⇒ $W = 11$ ⇒ invalid
- Try adding item three next ⇒ $W = 5$

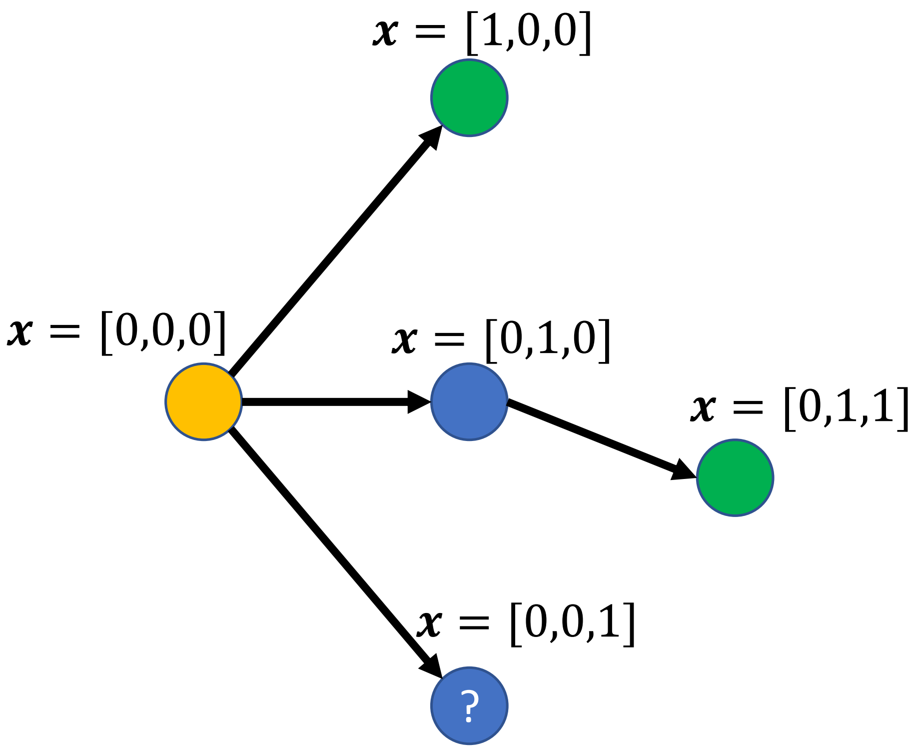
- We found a solution with $V = 2$
- Try adding the last item ⇒ $W = 14$ ⇒ invalid
- We discovered another final state, which is better than the previous one
- But it is the best state we can find?

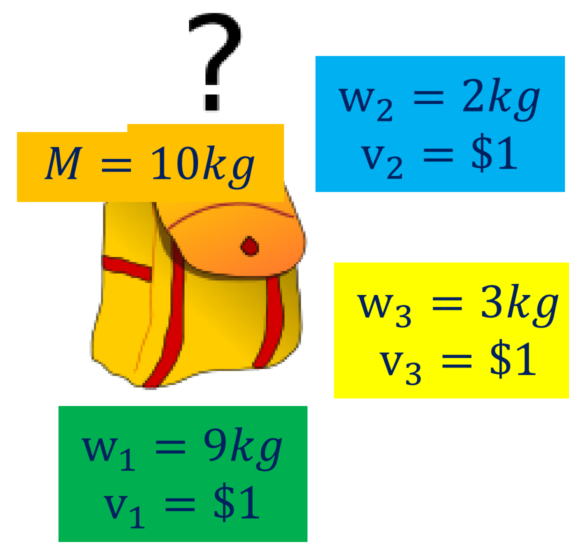




Example: Knapsack problem

To make sure we discovered the optimal solution, we need to backtrack again and select the third item first:

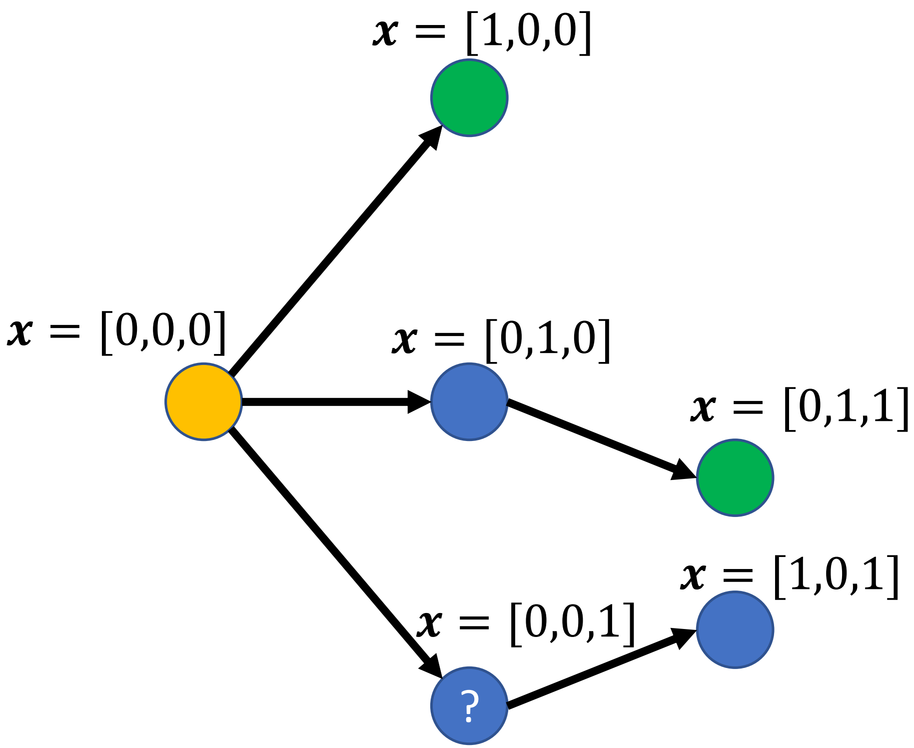


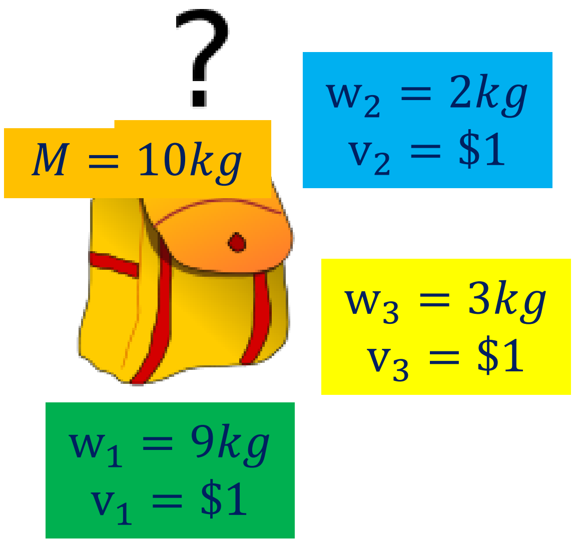


Example: Knapsack problem

To make sure we discovered the optimal solution, we need to backtrack again and select the third item first:

- Try to add item 1 now $\Rightarrow W = 12$

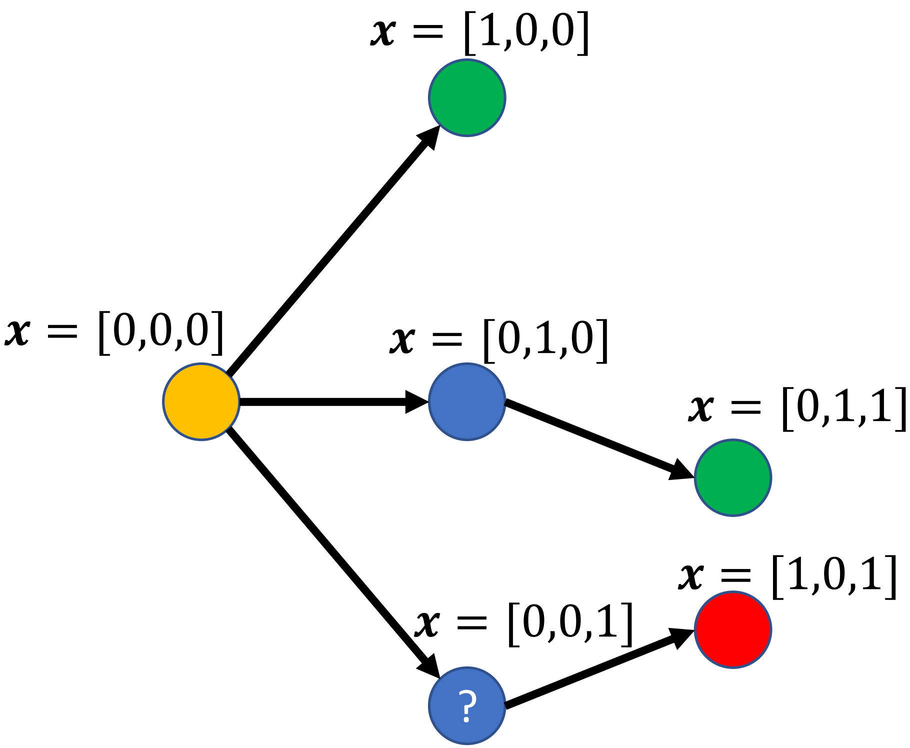


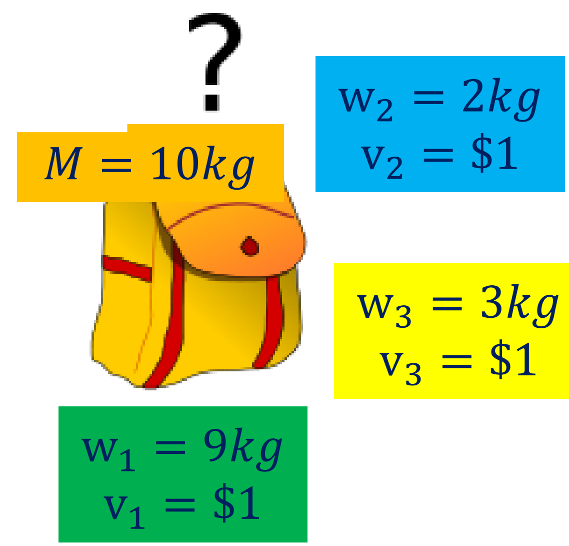


Example: Knapsack problem

To make sure we discovered the optimal solution, we need to backtrack again and select the third item first:

- Try to add item 1 now $\Rightarrow W = 12 \Rightarrow$ invalid

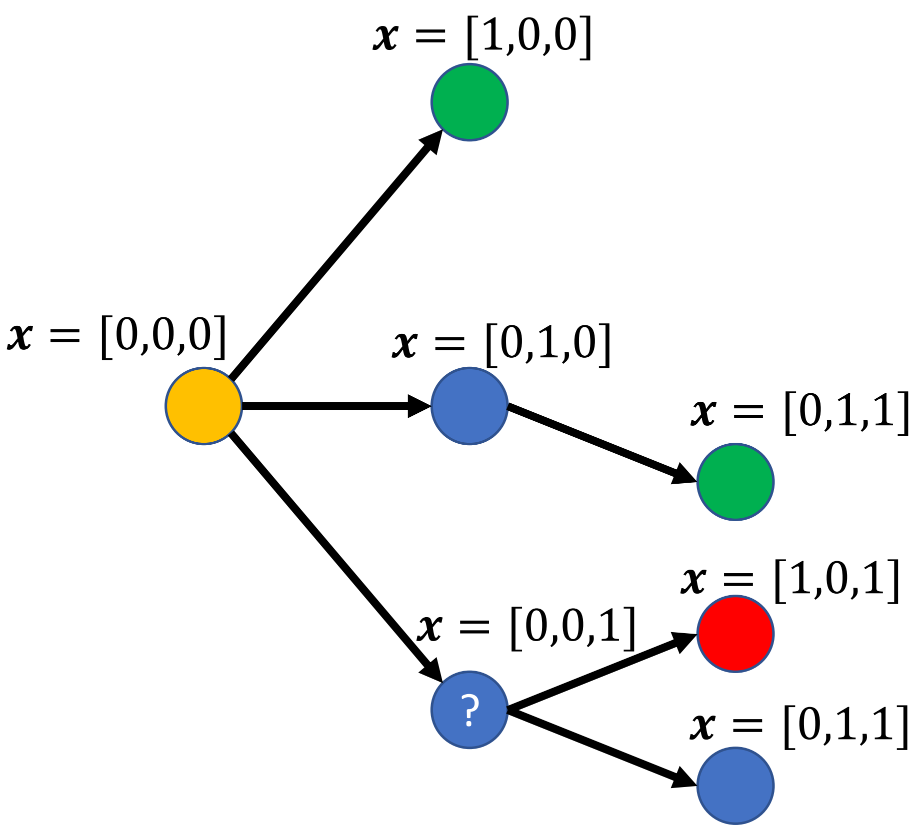


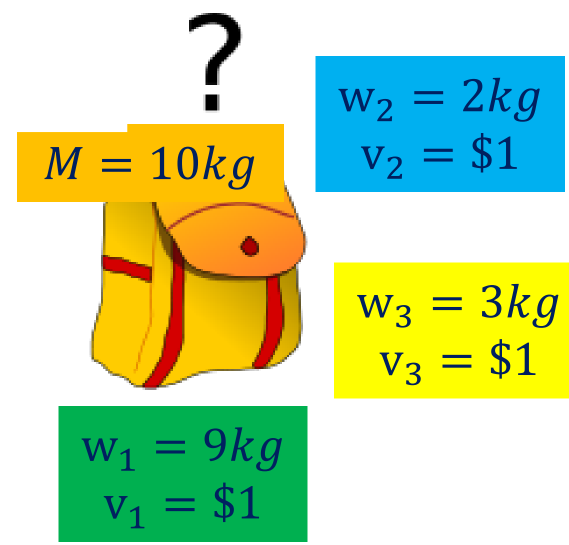


Example: Knapsack problem

To make sure we discovered the optimal solution, we need to backtrack again and select the third item first:

- Try to add item 1 now $\Rightarrow W = 12 \Rightarrow$ invalid
- Try to add item 2 now $\Rightarrow W = 5$



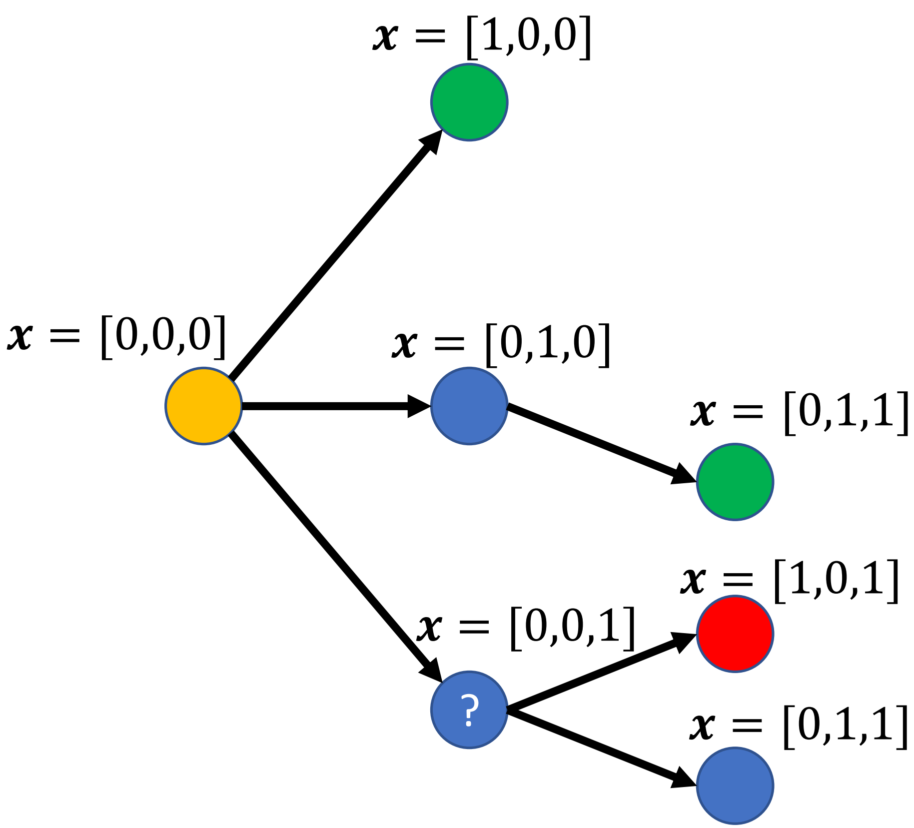


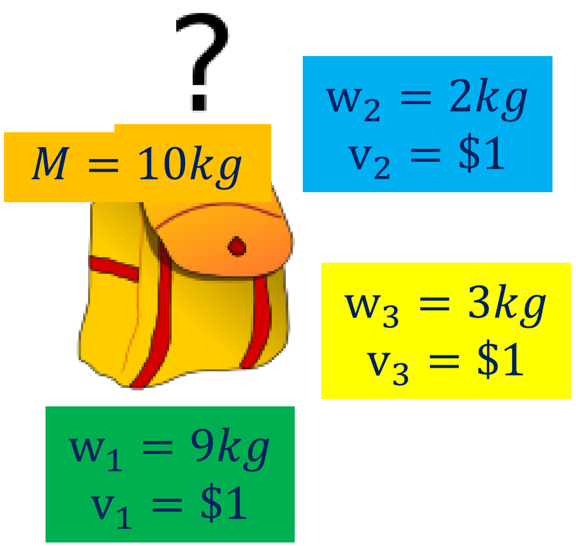
Example: Knapsack problem

To make sure we discovered the optimal solution, we need to backtrack again and select the third item first:

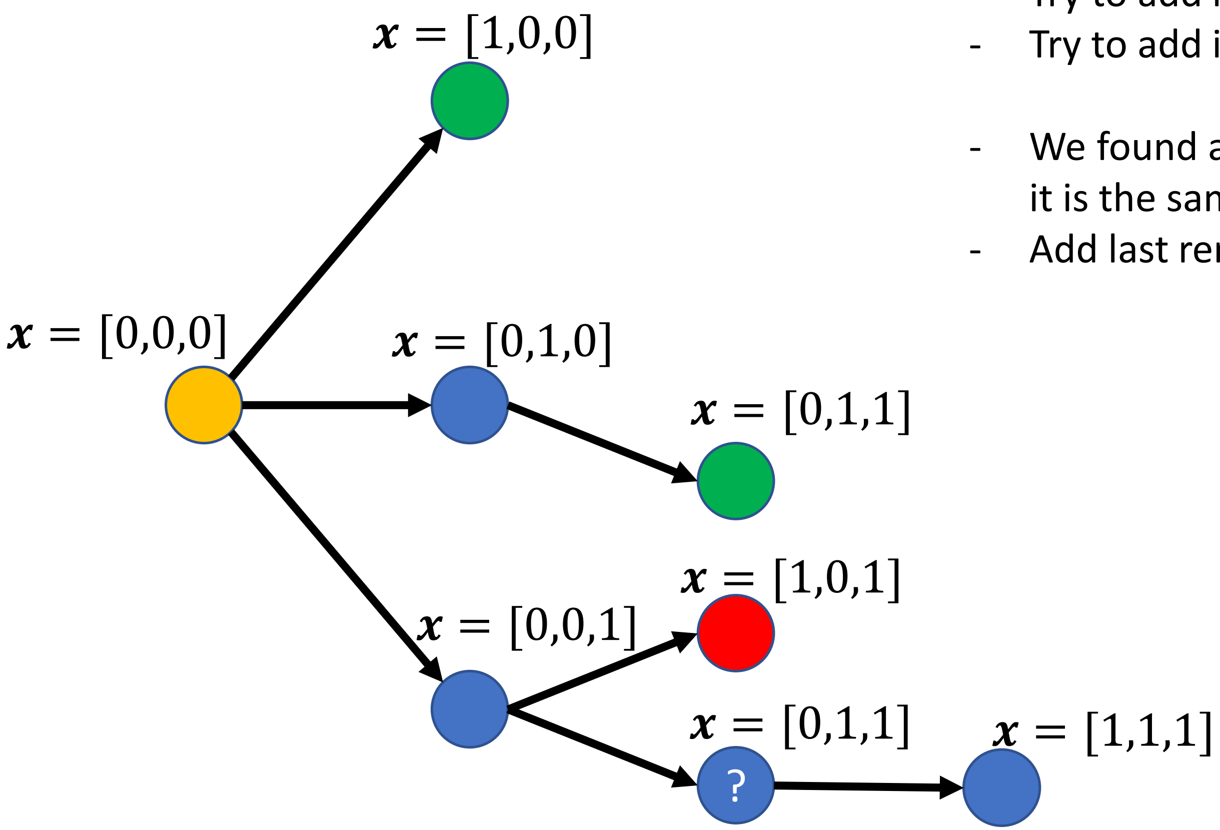
- Try to add item 1 now $\Rightarrow W = 12 \Rightarrow$ invalid
- Try to add item 2 now $\Rightarrow W = 5$

- We found another solution with $V = 2$ (actually it is the same as before)





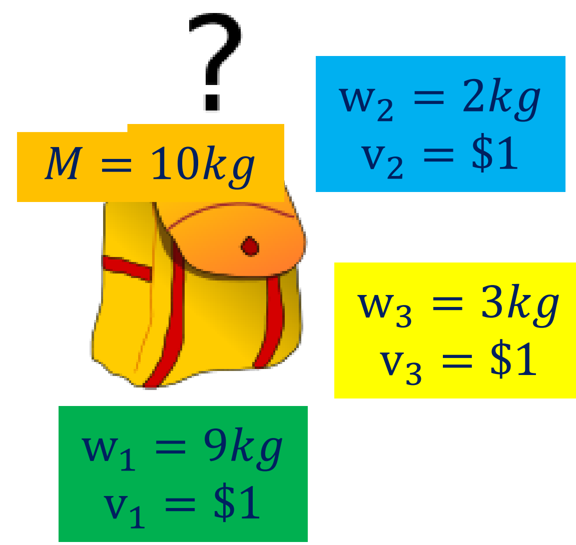
Example: Knapsack problem



To make sure we discovered the optimal solution, we need to backtrack again and select the third item first:

- Try to add item 1 now $\Rightarrow W = 12 \Rightarrow$ invalid
- Try to add item 2 now $\Rightarrow W = 5$

- We found another solution with $V = 2$ (actually it is the same as before)
- Add last remaining item $\Rightarrow W = 14$

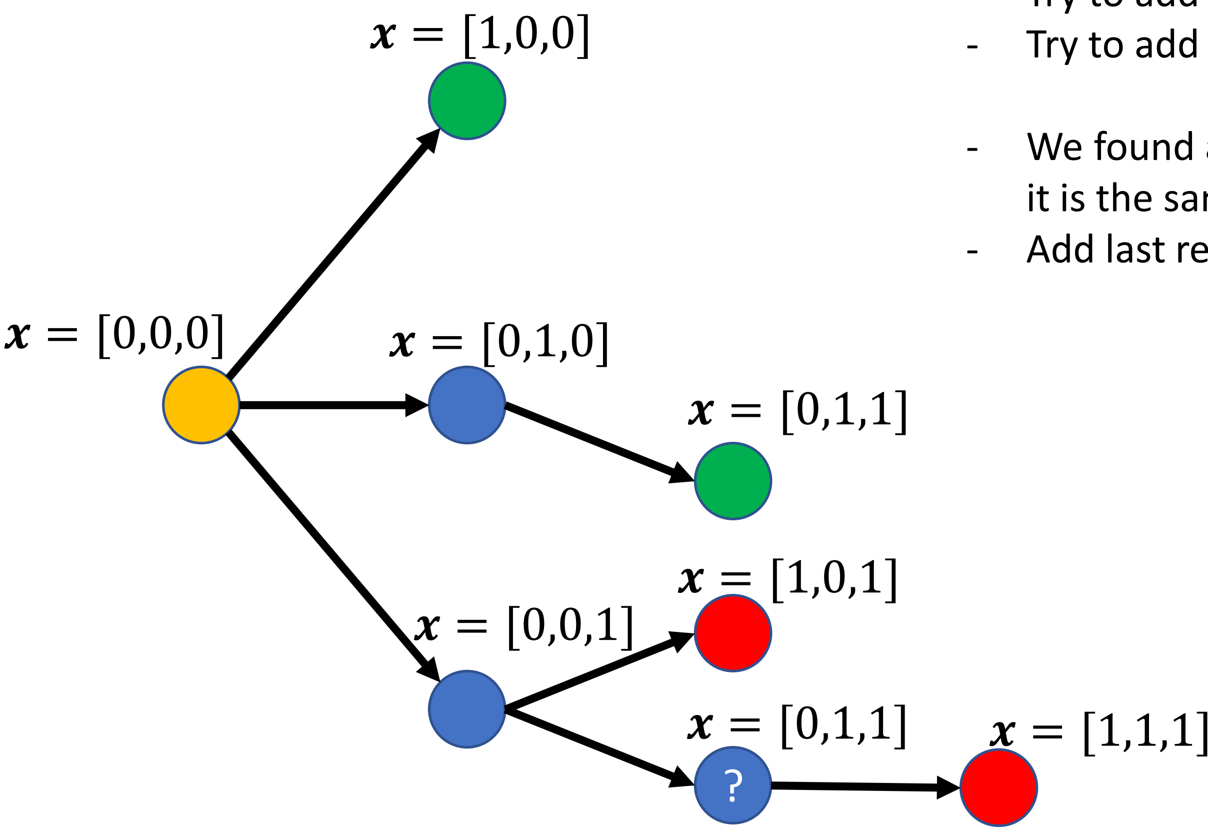


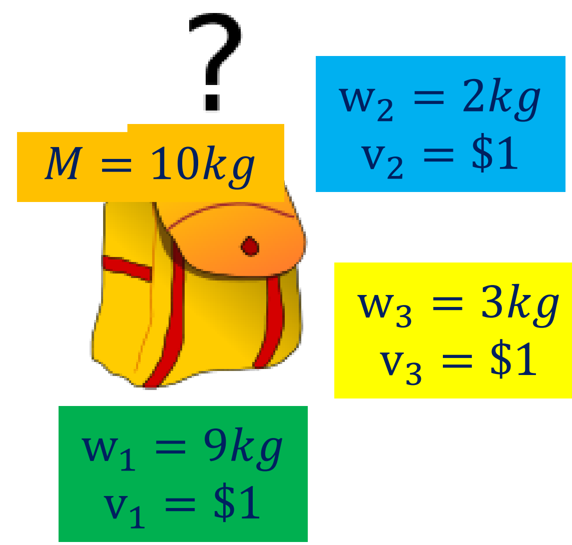
Example: Knapsack problem

To make sure we discovered the optimal solution, we need to backtrack again and select the third item first:

- Try to add item 1 now $\Rightarrow W = 12 \Rightarrow$ invalid
- Try to add item 2 now $\Rightarrow W = 5$

- We found another solution with $V = 2$ (actually it is the same as before)
- Add last remaining item $\Rightarrow W = 14 \Rightarrow$ invalid



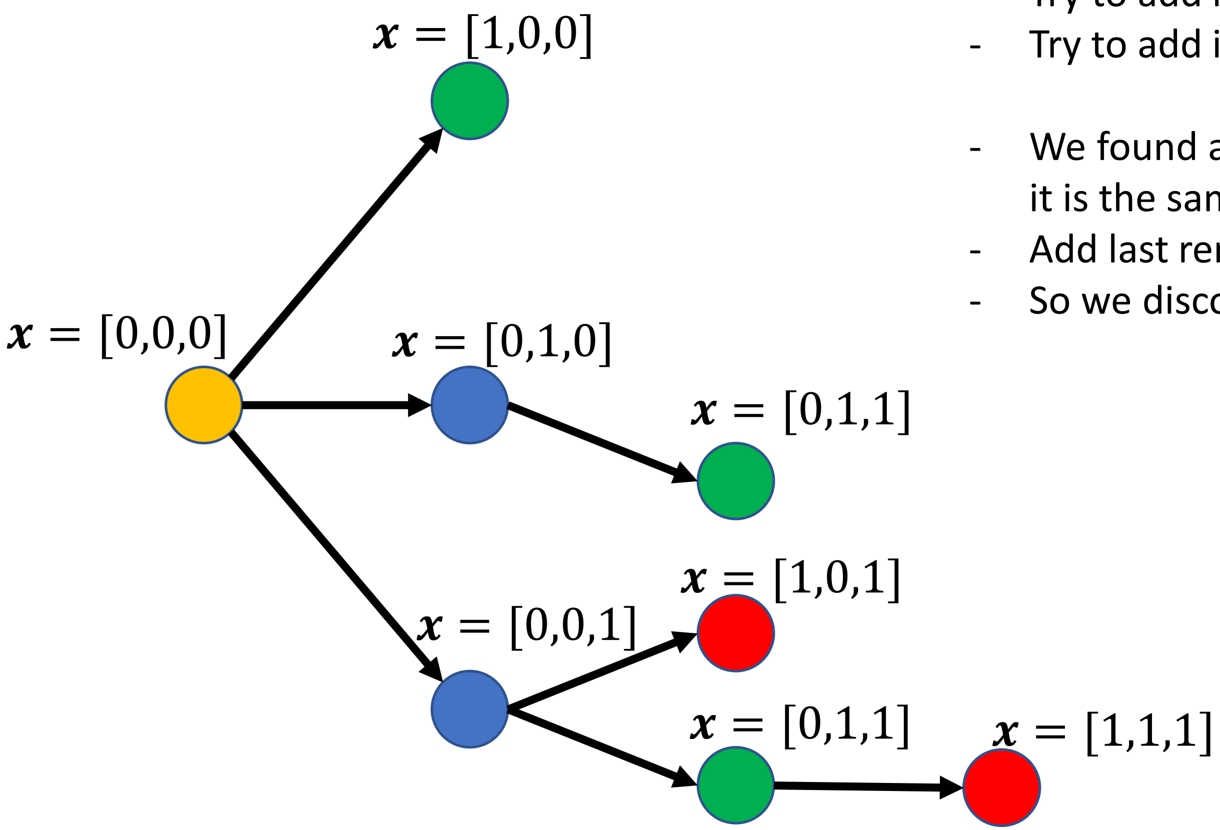


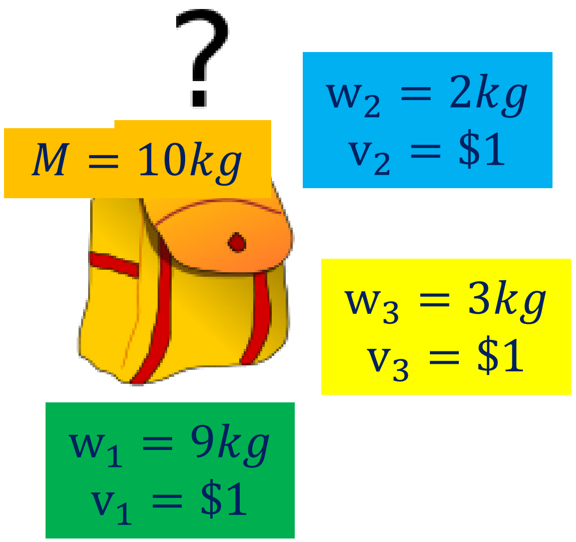
Example: Knapsack problem

To make sure we discovered the optimal solution, we need to backtrack again and select the third item first:

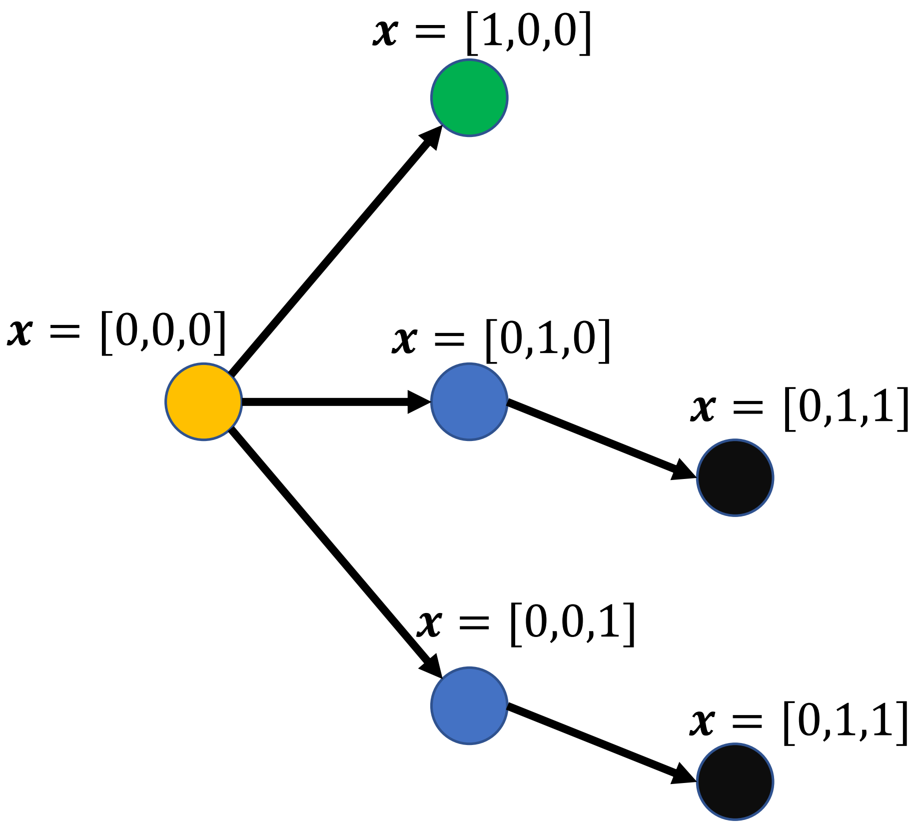
- Try to add item 1 now ⇒ $W = 12$ ⇒ invalid
- Try to add item 2 now ⇒ $W = 5$

- We found another solution with $V = 2$ (actually it is the same as before)
- Add last remaining item ⇒ $W = 14$ ⇒ invalid
- So we discovered another final state with $V = 2$





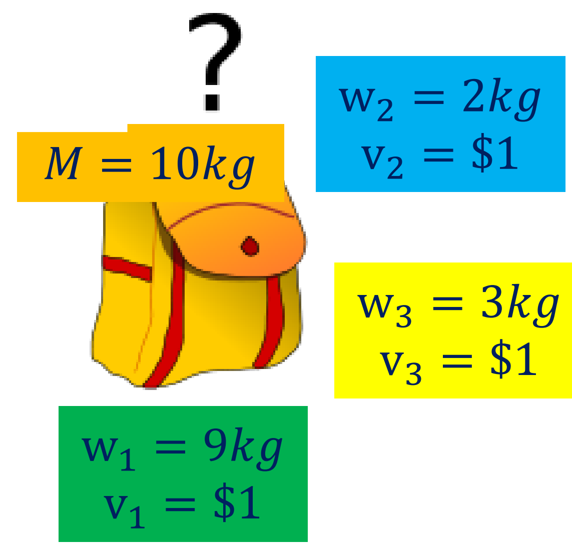
Example: Knapsack problem



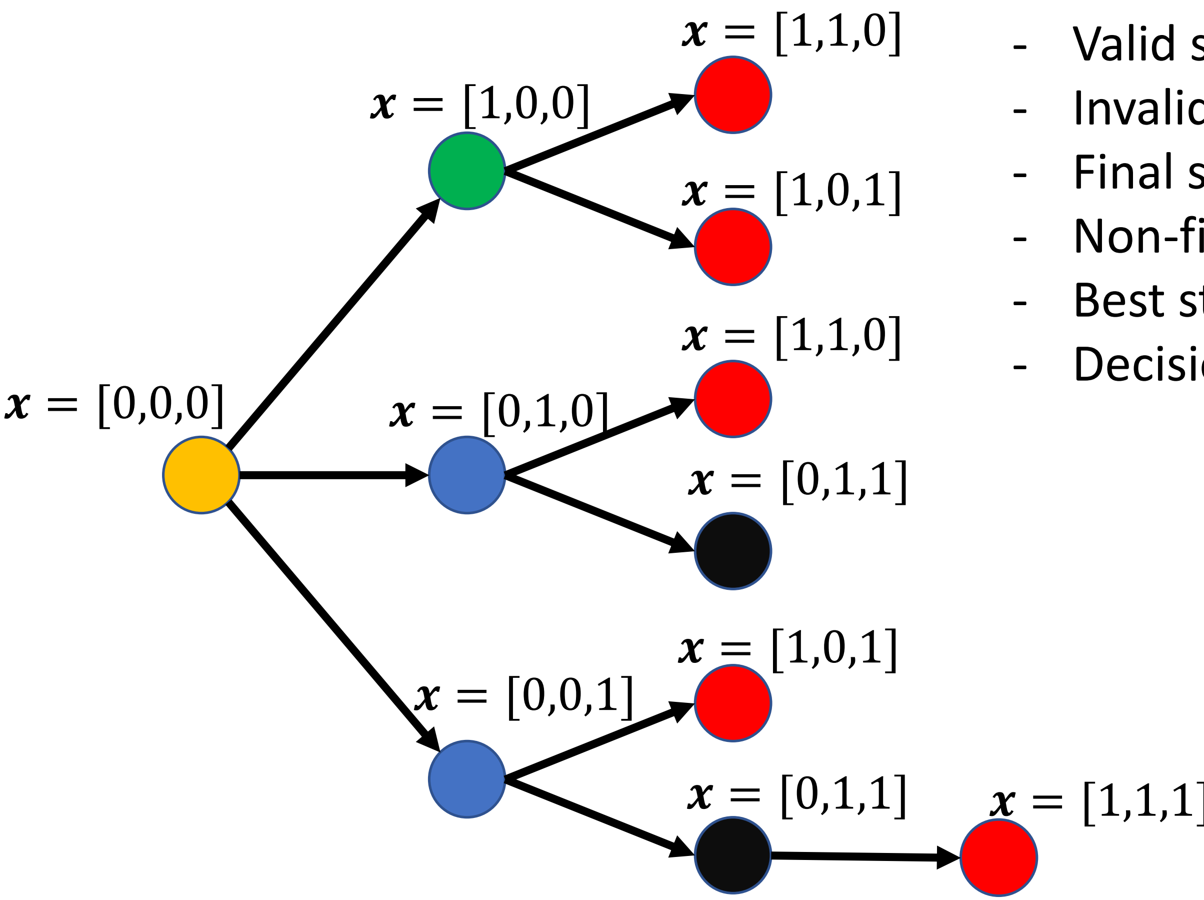
To make sure we discovered the optimal solution, we need to backtrack again and select the third item first:

- Try to add item 1 now $\Rightarrow W = 12 \Rightarrow$ invalid
- Try to add item 2 now $\Rightarrow W = 5$

- We found another solution with $V = 2$ (actually it is the same as before)
- Add last remaining item $\Rightarrow W = 14 \Rightarrow$ invalid
- So we discovered another final state with $V = 2$
- No more decisions left, so we found all optimal solutions!

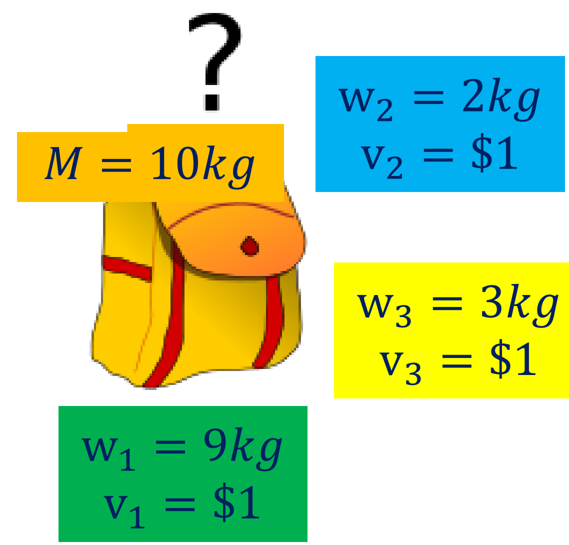


Example: Knapsack problem



We explored the following concepts:

- Initial state
- Valid state
- Invalid state
- Final state
- Non-final state
- Best state
- Decision making



Google OR Tools

- Google OR-Tools is open source software for combinatorial optimization, which seeks to find the best solution to a problem out of a very large set of possible solutions.

- It contains generic solvers and API interfaces for
  - Constraint Programming
  - Linear Programming
  - Mixed Integer Programming

- It also contains specialised solvers for
  - Routing
  - Packing
  - Min-Cut/Max-Flow
  - …

CP-SAT Solver

- The CP-SAT solver is used to solve constraint programming problems
- It can be used by importing:

```
from ortools.sat.python import cp_model
```

CP-SAT Solver

- We usually start by defining model variables

```
model = cp_model.CpModel()

x1 = model.NewIntVar(0,10,'x1')
x2 = model.NewIntVar(0,10,'x2')
b1 = model.NewBoolVar('b1')
```

First we declare the model

We then add variables to the model

Each variable needs a unique name

Integer variables are defined with a domain interval

CP-SAT Solver

- You can also define custom domains:

```
x3 = model.NewIntVarFromDomain(
                 cp_model.Domain.FromValues([1,3,5,7,9]))
```

- The CP-SAT solver is designed to work efficiently with integer (and bool) variables only
- In particular, it does not have a float variable type; but there are workarounds if absolutely necessary:
  - Multiply your variable with a large enough number to accommodate the necessary precision and work with that instead (e.g. use g instead of kg)
  - Use intervals to bound your solution, rather than trying to find the exact solution itself

CP-SAT Solver

- The next step is adding constraints

```
model.Add(x1 + x2 < 10)

product =  model.NewIntVar(0,100,'pr')
model.AddProdEquality(product, [x1,x2])
model.Add(product > 0)
```

The Add function adds “linear bounded constraints” only

More complex constraints might require additional variables

CP-SAT Solver

- There are many useful constraints, for instance:
  - AddAllDifferent(variables)
    To enforce that a set of variables all take on different values
  - AddBoolOr(literals)
    To implement a logical or for constraints
  - AddMinEquality(target, variables)
    AddMaxEquality(target, variables)
    To enforce that at least one value takes on a min/max
  - Etc.

CP-SAT Solver

- If we want to solve an optimisation problem, we can add an objective function

```
model.Maximize(x1 + x2)
```

or

```
model.Minimize(x1 + x2)
```

Like the Add function this has to be a linear and potentially requires additional variables

- This is optional, depending if we are interested in generating a feasible solution or if we want to find the best solutions

CP-SAT Solver

- Now that the model is defined we can solve it

```
solver = cp_model.CpSolver()

status = solver.Solve(model)
print(solver.StatusName(status))
```

First we create a solver instance

Solve the model by executing the solver

The solver returns a result status, which is one of: UNKNOWN, MODEL_INVALID, FEASIBLE, INFEASIBLE, OPTIMAL

CP-SAT Solver

- The solution variables can now be accessed through the solver as follows

```
print(solver.Value(x1))
print(solver.Value(x2))
```

- If the computation takes to long, it is sometimes useful to limit the computation time prior to executing the solver by setting its parameters

```
solver.parameters.max_time_in_seconds = 10.0
```

- Obviously we are not guaranteed the optimal solution, or indeed any feasible solution in this case

CP-SAT Solver

- If we want to see all feasible solutions we need to define a solution printer

```
class SolutionPrinter(cp_model.CpSolverSolutionCallback):
    def __init__(self, variables):
        cp_model.CpSolverSolutionCallback.__init__(self)
        self.variables_ = variables

    def OnSolutionCallback(self):
        print("Next solution:")
        for variable in self.variables_:
            print(self.Value(variable))
```

Every time the solver finds a valid solution it calls the OnSolutionCallback

Values of the current solution can be accessed using self.Value()

Class derived from CpSolverSolutionCallback

CP-SAT Solver

- Instead of calling the Solve function we call

```
solver = cp_model.CpSolver()
solver.SearchForAllSolutions(model, SolutionPrinter([x1,x2]))
```

- The model cannot have an objective function defined in this case

The Glop Linear Solver

- The Glop solver is used to solve linear programming problems
- It can be used by importing:

```
from ortools.linear_solver import pywraplp
```

- and then instantiating the wrapper

```
solver = pywraplp.Solver('LPWrapper',
                         pywraplp.Solver.GLOP_LINEAR_PROGRAMMING)
```

The Glop Linear Solver

- Again, we start by defining variables

```
x = solver.NumVar(0, solver.infinity(), 'x')
y = solver.NumVar(0, solver.infinity(), 'y')
```

A range of valid values needs to be defined

If a variable is unbound, you can use solver.infinity() to indicate this

Variables need a unique name

The Glop Linear Solver

- Next, we add linear constraints, for example $0 \leq x + 2y \leq 10$ is:

```
constraint1 = solver.Constraint(0, 10)
constraint1.SetCoefficient(x, 1)
constraint1.SetCoefficient(y, 2)
```

And a linear coefficient for each variable

We define the lower and upper bound of the constraint

The Glop Linear Solver

- We add a linear objective function, for example to maximise $3x + 4y$:

```
objective = solver.Objective()
objective.SetCoefficient(x, 3)
objective.SetCoefficient(y, 4)
objective.SetMaximization()
```

Indicator if the goal is to maximise of to minimise the objective function

A linear coefficient for each variable

The Glop Linear Solver

- Finally we execute the solver and retrieve the results

```
solver.Solve()

print (x.solution_value())
print (y.solution_value())
```

- If we need the value of the objective function, we have to compute it ourselves

```
obj = 3 * x.solution_value() + 4 * y.solution_value()
```

Thank you for your attention!